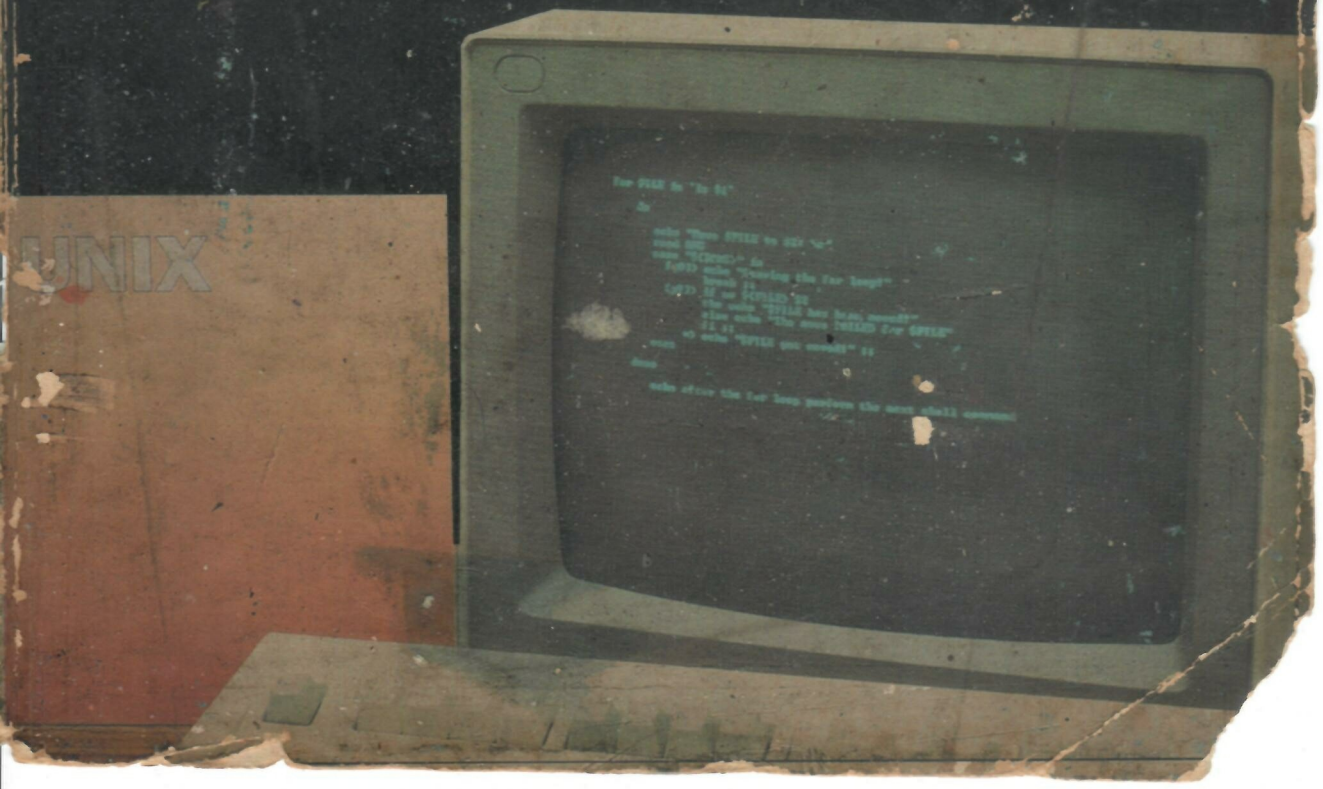
Illustrated

UNIX™ SYSTEM V

COMPLETE COMMAND REFERENCE WITH TUTORIAL



Computer & Electronics Books

Computer & Electronics Books

Illustrated UNIX System V/BSD

Robert Felps

BPB PUBLICATIONS
B - 14, CONNAUGHT PLACE, NEW DELHI-110001.

FIRST INDIAN EDITION-1992

Distributors:

BPB BOOK CENTRE
376, Old Lajpat Rai Market, Delhi–110006

BUSINESS PROMOTION BUREAU
8/1, Ritchie Street, Mount Road, Madras–600002

BUSINESS PROMOTION BUREAU
4-3-268-C, Giriraj Lane, Bank Street, Hyderabad–500195

COMPUTER BOOK CENTRE
12, Shrungar Complex, M. G. Road, Bangalore–560001

Copyright © Wordware Publishing, Inc.

No part of this book may be reproduced in any form or any means without permission in writing from Wordware publishing, Inc.

Published in india under arrangement with
WORDWARE PUBLISHING, INC., USA

UNIX is a registered trademark of UNIX System Laboratories, Inc.
PC-DOS is a registered trademark of International Business Machines Corporation.
MS-DOS and Xenix are registered trademarks of Microsoft Corporation.
VAX/VMS is a registered trademark of Digital Equipment Corporation.
Other poduct names mentioned are used for identification purposes only and may be trademarks of their respective companies.

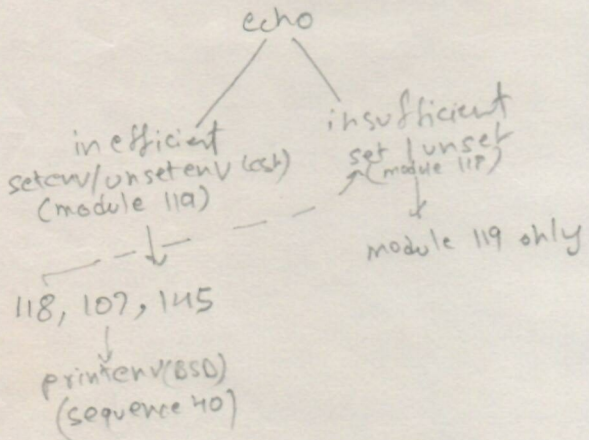
Published by Manish Jain for BPB Publications, B–14, Connaught Place, New Delhi and Printed by him at Pressworks, Delhi.

ACKNOWLEDGMENT

I would like to thank all of those involved with making this book a reality. Thanks to Nancy Fortson and Bill Kesler for hiring me into a UNIX environment that provided much of the knowledge to write this book. And to Nancy for being a patient mentor during my learning of advanced UNIX. Binion Amerson provided initial reviews of other documents which lead to the ultimate decision to write this book. Tina Pekrul provided initial ideas and direction on information to add to the book. Thanks to Bill Kesler and Scott Fort for being the best bosses I have worked for and allowing me to use computer resources to work on the book. And a special thanks to Chris Boyd for his effort in the final review of the book. And finally, I am especially grateful to Russ and Dianne Stultz for their willingness to publish my work. I would also like to thank Denise McEvoy for her effort in editing the book.

This book is dedicated to my wife for her support and encouragement during this long endeavor.

	module	Title	page
	7	banner (SV)	55
	38	echo	145
System V	38	setenv/unsetenv (csh)	490
Berkeley BSD	38	set/unset	484



Contents

Module	Title	Page
1	About This Book	1
2	A Sample Session with UNIX User Commands	6
3	System Overview	21
4	alias/unalias (csh, ksh)	29
5	apropos (BSD)	33
6	awk/nawk	35
7	banner (SV)	55
8	basename	57
9	bg (csh, ksh)	59
10	cal	61
11	calendar	63
12	cancel (SV)	65
13	cat	67
14	cd	71
15	chfn (BSD)	74
16	chgrp	76
17	chmod	79
18	chown (SV)	85
19	clear	87
20	cmp	88
21	col	90
22	comm	93
23	Commands	96
24	cp	103
25	cpio (SV)	105
26	crontab (SV)	111
27	crypt	115
28	csplit (SV)	118
29	cut (SV)	121
30	date	123
31	dd	126
32	df	130
33	diff	134
34	dircmp (SV)	139
35	dirname (SV)	141
36	dirs (csh)	142
37	du	143
38	echo	145
39	ed	148
40	egrep	160
41	env (SV)	164
42	eval	166
43	ex	168
44	exec	194
45	exit (csh, ksh, sh) / logout (csh)	196
46	expand/unexpand	199
47	export (sh, ksh)	201
48	expr	203
49	false	208

Contents (Cont.)

Module	Title	Page
50	fc (ksh) / ! (csh)	211
51	fg (csh, ksh)	221
52	fgrep	223
53	file	226
54	Filenames and Filename Generation	229
55	find	234
56	finger	239
57	fmt	242
58	fold (BSD)	245
59	getopts (ksh)	247
60	grep	251
61	groups	255
62	hash (ksh)	256
63	head	258
64	history (csh, ksh)	260
65	hostname (BSD)	263
66	id (SV)	264
67	Input/Output and Redirection	265
68	jobs (csh, ksh)	270
69	join	273
70	kill	276
71	ksh/csh/sh	280
72	leave (BSD)	295
73	let (ksh) / @ (csh)	297
74	line (SV)	302
75	listusers	304
76	ln	305
77	login	308
78	logname (SV)	312
79	lp (SV)	313
80	lpq (BSD)	318
81	lpr (BSD)	320
82	lprm (BSD)	323
83	lpstat (SV)	325
84	ls	329
85	machid (SV)	336
86	mail (BSD) / mailx (SV)	338
87	man (BSD)	363
88	mesg	367
89	mkdir	369
90	more/page	372
91	mv	377
92	newgrp (SV)	380
93	news (SV)	383
94	nice	385
95	nl (SV)	387
96	nohup	390
97	od	392
98	pack/unpack (SV)	395

Contents (Cont.)

Module	Title	Page
99	Parameters/Variables/Arrays	398
100	passwd	408
101	paste (SV)	412
102	pcat (SV)	414
103	pg (SV)	416
104	popd (csh)	421
105	pr	423
106	print (ksh)	428
107	printenv (BSD)	431
108	Programming Constructs - Shell Programming	432
109	ps	445
110	pushd (csh)	454
111	pwd	456
112	read (sh, ksh) / readonly (ksh) / set $< (csh)	458
113	Regular Expressions (REs)	461
114	rm	469
115	rmdir	472
116	script	474
117	sed	476
118	set/unset	484
119	setenv/unsetenv (csh)	490
120	shift	492
121	sleep	495
122	sort	497
123	spell	501
124	split	505
125	stop (csh, ksh)	507
126	stty	509
127	su	518
128	sum	521
129	suspend (csh, ksh)	523
130	tabs	525
131	tail	529
132	talk	531
133	tar	534
134	tee	539
135	test	541
136	time/timex	546
137	time (csh) / times (sh, ksh)	549
138	touch	552
139	tput (SV)	555
140	tr	559
141	trap (sh, ksh) / onintr (csh)	562
142	true	566
143	tset (BSD)	569
144	tty	573
145	typeset (ksh)	575
146	ulimit (ksh) / limit (csh)	579
147	umask	582

Contents (Cont.)

Module	Title	Page
148	uname (SV)	584
149	uniq	586
150	users (BSD)	588
151	vi/view	589
152	wait	614
153	wall (BSD)	616
154	wc	618
155	whatis (BSD)	620
156	whence (ksh) / type (sh)	622
157	whereis (BSD)	624
158	which (BSD)	626
159	who	628
160	whoami (BSD)	632
161	write	633
Appendix A	Terms and Definitions	637
Appendix B	UNIX Exercises	647
Appendix C	MS-DOS to UNIX Command Cross Reference	664
Appendix D	VMS to UNIX Command Cross Reference	665
Appendix E	File Formats	666
Appendix F	ASCII Collating Sequence	667
Appendix G	vi Quick Reference	674
Index		677

Recommended Learning Sequence for System V UNIX

Sequence	Title	Module	Page
1	About This Book	1	1
2	A Sample Session with UNIX User Commands	2	6
3	System Overview	3	21
4	ksh/csh/sh	71	280
5	Commands	23	96
6	Input/Output and Redirection	67	265
7	Parameters/Variables/Arrays	99	398
8	Filenames and Filename Generation	54	229
9	Regular Expressions (REs)	113	461
10	login	77	308
11	su	127	518
12	exit (csh, ksh, sh) / logout (csh)	45	196
13	newgrp (SV)	92	380
14	passwd	100	408
15	banner (SV)	7	55
16	echo	38	145
17	print (ksh)	106	428
18	cal	10	61
19	calendar	11	63
20	date	30	123
21	news (SV)	93	383
22	whence (ksh) / type (sh)	156	622
23	uname (SV)	148	584
24	machid (SV)	85	336
25	tty	144	573
26	id (SV)	66	264
27	logname (SV)	78	312
28	who	159	628
29	finger	56	239
30	listusers (SV)	75	304
31	groups	61	255
32	write	161	633
33	mesg	88	367
34	talk	132	531
35	export (sh, ksh)	47	201
36	env (SV)	41	164
37	set/unset	118	484
38	setenv/unsetenv (csh)	119	490
39	typeset (ksh)	145	575
40	ls	84	329
41	cd	14	71
42	pwd	111	456
43	pushd (csh)	110	454
44	popd (csh)	104	421
45	dirs (csh)	36	142
46	basename	8	57
47	dirname (SV)	35	141
48	mv	91	377

Recommended Learning Sequence for System V UNIX (Cont.)

Sequence	Title	Module	Page
49	ln	76	305
50	cp	24	103
51	rm	114	469
52	touch	138	552
53	file	53	226
54	cat	13	67
55	head	63	258
56	tail	131	529
57	pg (SV)	103	416
58	more/page	90	372
59	od	97	392
60	lp (SV)	79	313
61	cancel (SV)	12	65
62	lpstat (SV)	83	325
63	mkdir	89	369
64	rmdir	115	472
65	chown (SV)	18	85
66	chgrp	16	76
67	chmod	17	79
68	ulimit (ksh) / limit (csh)	146	579
69	umask	147	582
70	df	32	130
71	du	37	143
72	time/timex	136	546
73	time (csh) / times (sh, ksh)	137	549
74	tabs	130	525
75	clear	19	87
76	tput (SV)	139	555
77	sum	128	521
78	wc	154	618
79	cut (SV)	29	121
80	paste (SV)	101	412
81	cmp	20	88
82	comm	22	93
83	split	124	505
84	csplit (SV)	28	118
85	join	69	273
86	nl (SV)	95	387
87	col	21	90
88	fmt	57	242
89	fold	58	245
90	pr	105	423
91	diff	33	134
92	dircmp (SV)	34	139
93	crypt	27	115
94	pack/unpack (SV)	98	395
95	pcat (SV)	102	414
96	sort	122	497

Recommended Learning Sequence for System V UNIX (Cont.)

Sequence	Title	Module	Page
97	uniq	149	586
98	spell	123	501
99	tr	140	559
100	crontab (SV)	26	111
101	alias (csh) / unalias (ksh)	4	29
102	history (csh, ksh)	64	260
103	fc (ksh) / ! (csh)	50	211
104	hash (ksh)	62	256
105	bg (csh, ksh)	9	59
106	fg (csh, ksh)	51	221
107	jobs (csh, ksh)	68	270
108	suspend (csh, ksh)	129	523
109	stop (csh, ksh)	125	507
110	kill	70	276
111	nice	94	385
112	nohup	96	390
113	ps	109	445
114	tee	134	539
115	script	116	474
116	grep	60	251
117	egrep	40	160
118	fgrep	52	223
119	find	55	234
120	mail (BSD) / mailx (SV)	86	338
121	sed	117	476
122	ed	39	148
123	ex	43	168
124	vi/view	151	589
125	Programming Constructs - Shell Programming	108	432
126	let (ksh) / @ (csh)	73	297
127	shift	120	492
128	sleep	121	495
129	wait	152	614
130	true	142	566
131	false	49	208
132	test	135	541
133	trap (sh, ksh) / onintr (csh)	141	562
134	line (SV)	74	302
135	read (sh, ksh) / readonly (ksh) / set $< (csh)	112	458
136	eval	42	166
137	exec	44	194
138	expr	48	203
139	getopts (ksh)	59	247
140	awk/nawk	6	35
141	cpio (SV)	25	105
142	tar	133	534
143	dd	31	126
144	stty	126	509

Recommended Learning Sequence for Berkeley BSD

Sequence	Title	Module	Page
1	About This Book	1	1
2	A Sample Session with UNIX User Commands	2	6
3	System Overview	3	21
4	ksh/csh/sh	71	280
5	Commands	23	96
6	Input/Output and Redirection	67	265
7	Parameters/Variables/Arrays	99	398
8	Filenames and Filename Generation	54	229
9	Regular Expressions (REs)	113	461
10	login	77	308
11	su	127	518
12	exit (csh, ksh, sh) / logout (csh)	45	196
13	passwd	100	408
14	chfn (BSD)	15	74
15	echo	38	145
16	print (ksh)	106	428
17	cal	10	61
18	calendar	11	63
19	date	30	123
20	whence (ksh) / type (sh)	156	622
21	which (BSD)	158	626
22	whereis (BSD)	157	624
23	whatis (BSD)	155	620
24	man (BSD)	87	363
25	apropos (BSD)	5	33
26	hostname (BSD)	65	263
27	tty	144	573
28	who	159	628
29	finger	56	239
30	whoami(BSD)	160	632
31	users (BSD)	150	588
32	groups	61	255
33	write	161	633
34	wall (BSD)	153	616
35	mesg	88	367
36	talk	132	531
37	export (sh, ksh)	47	201
38	set/unset	118	484
39	setenv/unsetenv (csh)	119	490
40	printenv (BSD)	107	431
41	typeset (ksh)	145	575
42	ls	84	329
43	cd	14	71
44	pwd	111	456
45	pushd (csh)	110	454
46	popd (csh)	104	421
47	dirs (csh)	36	142
48	basename	8	57

Recommended Learning Sequence for Berkeley BSD (Cont.)

Sequence	Title	Module	Page
49	mv	91	377
50	ln	76	305
51	cp	24	103
52	rm	114	469
53	touch	138	552
54	file	53	226
55	cat	13	67
56	head	63	258
57	tail	131	529
58	more/page	90	372
59	od	97	392
60	lpr (BSD)	81	320
61	lprm (BSD)	82	323
62	lpq (BSD)	80	318
63	mkdir	89	369
64	rmdir	115	472
65	chgrp	16	76
66	chmod	17	79
67	ulimit (ksh) / limit (csh)	146	579
68	umask	147	582
69	df	32	130
70	du	37	143
71	time/timex	136	546
72	time (csh) / times (sh, ksh)	137	549
73	tabs	130	525
74	expand/unexpand (BSD)	46	199
75	tset (BSD)	143	569
76	clear	19	87
77	sum	128	521
78	wc	154	618
79	cmp	20	88
80	comm	22	93
81	split	124	505
82	join	69	273
83	col	21	90
84	fmt	57	242
85	fold	58	245
86	pr	105	423
87	diff	33	134
88	crypt	27	115
89	sort	122	497
90	uniq	149	586
91	spell	123	501
92	tr	140	559
93	alias / unalias (csh, ksh)	4	29
94	history (csh, ksh)	64	260
95	fc (ksh) / ! (csh)	50	211

Recommended Learning Sequence for Berkeley BSD (Cont.)

Sequence	Title	Module	Page
96	hash (ksh)	62	256
97	bg (csh, ksh)	9	59
98	fg (csh, ksh)	51	221
99	jobs (csh, ksh)	68	270
100	suspend (csh, ksh)	129	523
101	stop (csh, ksh)	125	507
102	kill	70	276
103	nice	94	385
104	nohup	96	390
105	ps	109	445
106	tee	134	539
107	script	116	474
108	grep	60	251
109	egrep	40	160
110	fgrep	52	223
111	find	55	234
112	mail (BSD) / mailx (SV)	86	338
113	leave (BSD)	72	295
114	sed	117	476
115	ed	39	148
116	ex	43	168
117	vi/view	151	589
118	Programming Constructs - Shell Programming	108	432
119	let (ksh) / @ (csh)	73	297
120	shift	120	492
121	sleep	121	495
122	wait	152	614
123	true	142	566
124	false	49	208
125	test	135	541
126	trap (sh, ksh) / onintr (csh)	141	562
127	read (sh, ksh) / readonly (ksh) / set $< (csh)	112	458
128	eval	42	166
129	exec	44	194
130	expr	48	203
131	getopts (ksh)	59	247
132	awk/nawk	6	35
133	tar	133	534
134	dd	31	126
135	stty	126	509

Module 1
ABOUT THIS BOOK

INTRODUCTION

This book describes the UNIX operating system user commands. It is based on AT&T UNIX System V Release 4 and Berkeley Software Distribution(BSD) 4.3. It provides you the knowledge and insight to fully use the power of the UNIX System.

You may ask what exactly is UNIX? UNIX is several things. It is a computer operating system. It is also a large set of utility programs. These utilities include general user, programmer, typesetting, communication, and administrative commands.

Because UNIX is a large computer operating system environment, only the more popular *User Commands* (over 160) will be discussed. An overview of the UNIX System, an MS-DOS to UNIX command cross reference, and a VMS to UNIX command cross reference are also included.

This book is designed for a broad range of users. It is for beginners who wish to learn UNIX from scratch. It provides intermediate and advanced users a complete reference containing command examples that are useful and provide insight on how to become a UNIX master. Finally, it serves the classroom instructor as an instructionally designed UNIX textbook.

The UNIX System is a sophisticated set of computer programs. The UNIX *kernel* is the main operating system program. The remainder of the System is made up of many powerful commands. The *User Commands* are available to all UNIX users.

The UNIX kernel (the operating system) is the software that controls and manages the computer system and the peripherals that are attached. The kernel controls the computer hardware, providing an interface between the application level software and the computer hardware. By managing the computer resources, the kernel can use a single computer and its peripherals to provide multiple users with the computer environment they need.

The UNIX *User Commands* provide the user with an extremely flexible and powerful computer environment. There are general service, file management, file processing, line printer (spooling), system status, intersystem communications, electronic messaging, pattern matching, text editing, command execution, and shell programming commands available. The most powerful and useful command is the UNIX shell. The Korn shell (**ksh**) is an enhanced version of the Bourne shell (**sh**). It is used in *Illustrated UNIX* as the default UNIX shell.

The **ksh** is a sophisticated command interpreter. It does not display user friendly menus. Instead, it displays a *prompt* and a *cursor*, requiring you to enter your command from the keyboard. A command is the instruction you type on your keyboard that informs **ksh** what program to execute. To a new user, **ksh** may appear to be intimidating because of its terse and cryptic commands and syntax. It is important and necessary that you learn and understand how **ksh** works. Most users think of **ksh** (or some other shell) as UNIX. This is actually a valid assumption since you interact with **ksh** and not the operating system.

ORGANIZATION

This book is organized into easy-to-read, example-packed *modules*, to fit the broad range of users it addresses. These modules contain Description, Applications and Typical Operation sections.

Because this book covers two versions of the UNIX operating system, it is necessary to distinguish between commands supported by one version and not the other. This problem also arises with different types of shells (command interpreters).

Both versions of the operating system support the Bourne shell (**sh**). The SV version supports an enhanced shell, the Korn shell (**ksh**). The BSD operating system supports the C shell (**csh**), a more interactive oriented shell. The ksh is the standard shell on SV. It can be purchased for most BSD systems.

If a command is only supported by one of the shells, then the module is marked accordingly. If the module is only usable if you have the **csh**, the module is marked as such. The same is true of modules only usable if the **ksh** is available.

If a command is only found on one system, i.e., System V or BSD, the entire module is specified as that type of module. System V modules are marked as **SV** and Berkeley modules are marked as **BSD**.

If a command is shared by both systems, then the base information in the module is System V with BSD text interventions. The inserts of BSD related material are clearly marked. For example,

---

BSD (Berkeley)

This text only appears where BSD information applies.

Options	Under the Options section the options discussed in the BSD section are either in addition to the SV options, perform a different function, or are specifically marked as unsupported.

This probably seems very confusing. The following table helps clarify how commands (modules) are marked and why.

Command Mark	*Description*
SV	The external command is only supported on System V systems.
BSD	The external command is only supported on BSD systems.
csh	The internal command is only supported by the **csh**.
ksh	The internal command is only supported by the **ksh**.
internal	The command is an internal shell command. It is performed by the shell.
external	The command is external from the shell. It is a separate program that executes apart from the shell.
csh/ksh	The command is supported by the **csh** and **ksh** but not the **sh**.
“”(no marks)	The command is supported by SV and BSD if external.
“”(no marks)	The command is supported by all shells.
A(X-csh)	The letter as used by the ksh and the letter X used by the csh.
A(X-bsd)	The letter as used by SV and the letter X used by BSD.

NOTE

The BSD operating system is distributed by the University of California. Vendors purchase the source code and make proprietary changes as they desire. Thus the BSD commands may or may not correspond to the documentation contained in this book. An AT&T System V V32 license is required to purchase the BSD source.

NOTE

A module marked as SV only or BSD only may actually be found on different type systems. This is usually because the vendor starts with a System V base and includes BSD extensions or vice versa.

Two learning sequences are provided at the beginning of this book. One is a list of all System V modules, the other is for Berkeley (BSD) UNIX. Related commands are grouped together. These learning sequences are extremely useful to chart your progress if you are learning or teaching UNIX *User Commands*.

When using the learning sequence it is best to decide which shell you are going to use. If you have the **ksh**, use it. If you don't have the **ksh** but do have the **csh**, then use it. Otherwise suffer with the **sh**. Once you select a shell, continue using it throughout the book. If a module is specifically designated for a shell you are not using, skip to the end of the module and find the next module to start using.

This module provides a brief overview of this book. It provides information about using the book, its contents, and how it is organized. It informs you about hardware and software requirements. It also tells you what you should know before continuing in the learning sequence.

Module 2 is a sample session using UNIX. It is designed for the user who wants to dive in immediately and for the person who needs to step through a simple exercise using the UNIX System. Module 2 takes you on a tour of the UNIX System, from logging in and performing tasks with UNIX commands to logging out. As you follow along on your computer, you will discover how easy it is to use the power of UNIX.

Module 3 is a brief overview of the UNIX System. It provides valuable insight into the features and functionality of UNIX. It must be read to understand how UNIX works and how you can better use the resources that UNIX provides.

Modules 4 through 166 describe and illustrate the *User Commands*. These modules are arranged in alphabetical order for easy reference, once you have finished the learning sequence.

Appendix A contains a list of terms and definitions that are used in this book. This appendix is a useful tool for clarifying words that were not defined when you first encountered them in the text or that are not a module within themselves.

Appendix B contains UNIX exercises. It is provided for the classroom and self-teaching situations. If you are a classroom instructor, you may wish to include these exercises in student assignments. If you are learning UNIX on your own, the exercises are a good way to see what you've learned about a command. If you can answer the questions, you are ready to move on to the next module in the learning sequence.

Appendix C contains an MS-DOS to UNIX command cross reference. It provides a table of MS-DOS commands and the closest UNIX equivalent.

Appendix D contains a VMS to UNIX command cross reference. It contains a table of VMS commands and the closest UNIX equivalent.

Appendix E contains file formats for the password and group files.

Appendix F contains the ASCII collating sequence used by UNIX commands to distinguish the proper order (precedence) of bytes.

Appendix G contains a quick reference for the **vi** full screen text editor.

HARDWARE AND SOFTWARE REQUIREMENTS

The UNIX System has been ported (rewritten to run on different hardware) to every major hardware vendor's machine. With over 150 designs of computers running UNIX, ranging from micros to supercomputers, you will discover differences between the various UNIX implementations. Hopefully, this book provides a standard resource of information that will allow you to be productive on your UNIX System.

INITIAL REQUIREMENTS Because UNIX is a multitasking, multiuser system, it is hard to know each system's requirements. Therefore, the following list is a general guideline for what you as a user will need before you can begin to work on UNIX. You will have to consult your UNIX system administrator to establish some of these requirements.

1. An account on the UNIX System; provides you with a user name. In this book **mylogin** has been used as the user name. Substitute your user name in place of **mylogin** as you work the examples in this book.
2. An ASCII terminal; must be capable of FULL-DUPLEX mode.
3. A direct line to the computer from your terminal or a modem connected to your terminal.

Request that your system administrator *E-mail* you a short letter when your account is set up.

The key to using UNIX is knowing what it has to offer. It is the most flexible and powerful general purpose operating system on the market, and as a user you can accomplish many tasks with little effort by using it efficiently. This book provides the necessary information for you to use the system.

WHAT YOU SHOULD KNOW

As a minimum you should be familiar with ASCII terminal turn-on and setup. If you have a printer connected to your UNIX System, you should have some general knowledge about its page and print size. If you are using a modem to connect to your UNIX System, it is advisable to learn how to operate it.

Headings within each module are **BOLDFACE** and uppercase or *ITALICS* and uppercase. All **boldface** strings are literals that you are to type as they appear. Some of these are examples, requiring files that may not exist. *Italic* print usually specifies strings you are supposed to substitute in commands or the names of files or special phrases.

You must know your terminal type to use some of the UNIX commands. You may have to ask your UNIX system administrator or some other experienced UNIX user.

Finally, you should be familiar with your keyboard. If the keys on your keyboard have special symbols instead of words, use the following table to see how they are referred to in this book.

Key Name	*Character Sent*	*Keyboard Symbol*
Return	Ctrl-M	Return, Carriage Return, or ↵
New-line	Ctrl-J	Line-Feed
Backspace	Ctrl-H	←
Tab	Ctrl-I	← →
Shift	Capitals/Symbols	Shift, ⇑

ENVIRONMENTAL ASSUMPTIONS

You will need to have the system administrator set up your account and answer the following questions. Many system administrators will provide this information to you without asking. The defaults used in this book are given in parentheses.

1. What is my user name or login name? (mylogin)
2. What is my password? (imauser2)
3. Which key transmits a command to the system? (Return)
4. Which key will erase a character? (# or Ctrl-H or DEL)
5. Which key will kill an input line? (@ or Ctrl-X or Ctrl-U)
6. Which key interrupts program execution? (Del or Ctrl-C)
7. What is the *termcap* or *terminfo* name of my terminal? (vt100)
8. Which shell will I be using? (ksh, i.e., the Korn shell) The notation of the shell, **ksh**, used throughout this book are references to the Korn shell. The Korn shell is a super-set of the standard Bourne shell (**sh**). Most BSD systems supply the **sh** and **csh**. The version of the **sh** on BSD is usually ancient, UNIX Version 7.
9. What are the start/stop characters for output control? (Ctrl-Q/Ctrl-S)

The following list of assumptions have been made about the UNIX System and the logins used throughout this book. Please make the appropriate substitutions as you use the examples in this book.

1. The name of the UNIX System is **cj** (casper junior).
2. The HOME directory for **mylogin** is **/u1/ts/mylogin**.
3. The user ID for **mylogin** is 101, the group ID is 40.
4. Various other user names are used throughout the book. Substitute valid user names in place of these names when necessary.
5. The prompt (PS1) for *mylogin* is **cj>**, the secondary prompt (PS2) is **cj>?**.
6. Turn to Module 2 to continue the learning sequence.

Module 2
A SAMPLE SESSION WITH UNIX USER COMMANDS

DESCRIPTION

This module provides basic information about getting started using *UNIX*. It then leads you on a "hands-on" tour of the most commonly used UNIX commands. Because this module is intended to get you familiar with UNIX, it does not provide in-depth explanations. If you wish to know more about a command while you are using it, refer to the module for the given command. The third module provides additional information about features of the UNIX operating system.

The following is a brief outline of information contained in this module. The information discussed in the first four sections is usually set by the system administrator when your login account is created.

- Terminal setup
- Communications between UNIX and your terminal
- Logging in to the UNIX System
- Correcting typing mistakes
- Setting/changing your password
- Knowing which version of UNIX you are using
- Knowing which shell you are using
- UNIX environment
- Executing commands
- Common commands
- Text editing with vi/ex
- Communicating with others
- Logging out of the UNIX System

BEFORE YOU START

Before you can begin using the UNIX operating system you must have a login account. You should request the login account from the system administrator. If you do not have a system administrator, you must create a login account yourself. Check your System Administrator's Guide or Reference Manual for instructions on how to do this.

You will also need a terminal connected to your system. There are several ways for terminals to communicate with the UNIX System: via direct wire, modem, or terminal servers. Your system administrator should connect your terminal to the system.

LOGGING IN TO THE UNIX SYSTEM

Once you have a "login:" prompt on your terminal and a login account, you can log in to the UNIX System. Throughout this book we assume /u1/ts/mylogin is your login (HOME) directory. After you log in and know your HOME directory substitute it in place of /u1/ts/mylogin. We also use the "mylogin" login name; you should replace it with your login name.

1. At the *login:* prompt type **mylogin** and press **Return**.

```
cj login: mylogin
```

TIP: Make sure your CAPS key is not activated. If you have a *LOGIN:* prompt in all capital letters, press **Ctrl-D** and wait for a new *login:* prompt to appear, then try to log in to the system.

NOTE

Your login name must contain one lowercase character. If you do not type at least one lowercase character, UNIX assumes your terminal cannot generate lowercase ASCII characters and treats all characters as uppercase for the remainder of your login session.

2. Type **iamuser2** and press **Return** at the passwd: prompt to log in to the system. Be patient! Depending on your system and its load factor, it could take from a few seconds to a few minutes for the system to respond.

```
cj login: mylogin
passwd:
```

NOTE

Your password will not be printed, or "echoed," as you type it. The UNIX System disables the output so nosy people are not able to read your password.

3. If your login was successful, you should see information displayed on your screen. If your login attempt was unsuccessful, your display will resemble the following,

```
cj login: mylogin
passwd:
Login incorrect.
login:
```

NOTE

If, after a couple of attempts, you cannot get logged in, check your login name and password with the ones your system administrator gave you. If you are still unable to gain access into the system, contact your system administrator, he/she likes to hear from frustrated users!

Messages from the system The following list is a possible sequence of what may appear on your screen once you type the correct password. You may be required to type information to complete the login sequence. This depends on your local system.

- General information about the local UNIX System is displayed on your terminal from the message of the day file (/etc/motd).
- System displays or lists unread news items.
- System displays or informs you of mail you have received.

- You may be requested to set your terminal type.
- System displays various other information the system administrator has set up.
- A prompt from the UNIX shell (command interpreter) appears. The default prompt is a dollar sign ($).

C Shell

- The default prompt for the csh is a percent sign (%).

CORRECTING TYPING MISTAKES

The UNIX System reads each individual character and temporarily saves or buffers them until you press Return or Line Feed. It then interprets the entire line and performs the requested operation.

The UNIX System performs full duplex input/output (I/O) with your terminal. This means you can type characters in to UNIX while UNIX is displaying information out to your terminal. This will result in a messy, hard-to-read screen, but the data will flow both directions, and the input flow will be interpreted in the correct sequence. You may also type ahead of the UNIX System's ability to read the characters, referred to as UNIX's read-ahead feature. The number of read-ahead characters you may type is limited but usually it is more than you will ever need (256 to 4096).

Default Correction Keys The @(^U-csh), #(DEL-csh) and \ characters have special meaning to UNIX. The @(^U-csh) character deletes the current input line and is referred to as the line kill character. The #(DEL-csh) erases the last character and is referred to as the erase or backspace character. Multiple #(DEL-csh) erases back to the beginning of the current line. The \ character escapes the meaning of any special character. Therefore \# prints a # and does not perform a backspace.

Resetting Correction Keys The @(^U-csh) and #(DEL-csh) keys can easily be reset. The standard reset values are Ctrl-X (while pressing and holding the Ctrl key, press the X key) used in place of @, and Ctrl-H (Backspace) used instead of #. Most csh users (who are usually on BSD systems) do not reset the ^U and DEL keys.

1. Display the current settings that UNIX interprets as special keys on your terminal by typing **stty** and pressing **Return**. Notice the erase and kill default settings. Depending on your system administrator, your settings may differ. The output format for **stty** may differ if you are on a BSD system.

```
cj> stty
speed 38400 baud; -parity cread
erase = #; kill = @; swtch = ^`;
-inpck icrnl -ixon onlcr tab3
echo echoe echok
```

BSD (Berkeley)

cj> stty everything

---

2. Redefine the erase and kill characters by typing **stty erase "^h" kill "^x"** and pressing **Return**. The "^" (caret) is typed as Shift-6; do not type Ctrl-H. Notice **stty** does not display a response.
3. Display the new settings by typing **stty** and pressing **Return**.

```
cj> stty
speed 38400 baud; -parity cread
erase = ^h; kill = ^x; swtch = ^`;
-inpck icrnl -ixon onlcr tab3
echo echoe echok
```

4. Type **echo Ti#his is a #n example.** at your shell prompt and press **Return**. Use the appropriate erase key (Ctrl-H/Backspace) in place of the # keys. Notice that when you press your Backspace key the cursor moves back one space.
5. Type **echo Another eamxpl@echo Another example** and press **Return**. Use the appropriate line kill key (Ctrl-X) in place of the @ key. Notice that when you press your kill key the cursor jumps to the beginning of the next line; the old line is ignored and a new line of input is read.

BSD (Berkeley)

The new BSD device driver will erase the current line and move your cursor to the beginning of the current line.

---

6. Type **cat /etc/group** and press **Return**. Be ready to immediately press **Ctrl-S** to stop the output. The Ctrl-S does not appear on your terminal. To resume scrolling of the output press **Ctrl-Q**.
7. If you have I/O problems with your terminal, check the setting with **stty** and refer to the **stty** command (Module 126) for valid parameters. Don't hesitate to request help from a knowledgable source about your terminal and the tty device driver. This is usually a difficult part of UNIX to get a handle on.

KNOWING WHICH UNIX YOU ARE USING To know which type or version of UNIX you are using, use the **uname** command. Type **uname** and press **Return**. If the name of your system is displayed, you are probably on System V UNIX. If the shell returns a message like "uname: command not found," you are probably on a BSD system.

Another check to perform is to type **ls /usr/ucb** and press **Return**. If a listing is returned, you are probably on a BSD-type system. If a message like "/usr/ucb not found" is returned, you are probably on a System V system.

These are not guaranteed checks but they are fairly reliable. If both return info then you are on a hybrid SV and BSD system.

KNOWING WHICH SHELL YOU ARE USING To know which shell you are using, use the **ps** command.

On System V type **ps** and press **Return**.

BSD (Berkeley)

On BSD type **ps -u** and press **Return**.

The right-hand column contains the names of the programs currently executing. If one is **csh**, you are using the C shell; **sh** is the Bourne shell, and **ksh** is the Korn shell.

Another alternative is to type **echo $0** and press **Return**. If you are using the **sh** or **ksh**, the corresponding string is displayed on your terminal. If you are using the **csh**, nothing is displayed.

UNIX ENVIRONMENT

The UNIX shell maintains an *environment* that controls how commands are interpreted and performed. A few of the more common environment *variables* and concepts are presented throughout the following paragraphs.

INPUT/OUPUT (I/O) The UNIX System handles I/O as streams of data. The streams flow into and out of commands, files, or devices. The shell has three default types of I/O. The input stream is referred to as *standard input* and is usually generated by your terminal keyboard. The output stream is referred to as *standard output* and is usually displayed to your terminal screen. *Standard error* is error output from programs which also defaults to your terminal screen.

Standard Input/Output/Error

Conventional Names	*Default Device Input or Output is Connected to*	*File Descriptor (Used by the Shell)*
Standard Input	Terminal Keyboard (or from a pipe of file)	0
Standard Output	Terminal Screen (or to a pipe or file)	1
Standard Error	Terminal Screen (or to a pipe or file)	2

REDIRECTION The standard I/O can be redirected from your terminal to a file or a device. You can store data in files or send the contents to a command as input. Output from one command can be redirected to the input of another command. This is called "piping."

PATH NAMES Pathnames are used to refer to directories and files not in the present working directory. A path is made up of a "/" followed by a subdirectory name or a filename, multiple "/" and subdirectories can be listed. *Full* pathnames begin with a "/", which denotes the root directory. Relative pathnames do not begin with a "/". Instead they start with a directory name or ".". There are several types of relative pahtnames. Subdirectory paths begin with the subdirectory's name, not a "/". Indirect paths begin with a "..", which allows dynamic backward movement within the file system.

1. Change directories using a full pathname by typing **cd /usr/bin** and pressing **Return**.
2. Display the contents of the /usr/bin directory by typing **ls -x** and pressing **Return**. Your output may vary depending on the type of UNIX System you are using, but it should resemble the following.

```
cj> ls -x
300         300s      3bnstat   4014      450       Oar
```

```
admin      asa      at       awk      b16as    b16cc
b16ckfile  b16conv  b16cprs  b16dis   b16dump  b16ld
.
.
.
```

BSD (Berkeley)

The output for BSD systems is slightly different.

---

3. Return to your home directory by typing **cd** and pressing **Return**.

PRESENT WORKING DIRECTORY The *present working directory* is often referred to as the *current working directory* or *current directory*. Since the command that returns the present working directory is "pwd," we will refer to it as such. The UNIX System has file systems that are hierarchical tree structures containing directories and files.

1. Display the present working directory by typing **pwd** and pressing **Return**.
2. To change the present working directory type **cd ..** and press **Return**. Notice that no response is given if the command worked.
3. Display the present working directory by typing **pwd** and pressing **Return**. Notice you have moved up the directory path one position. The ".." refers to the previous or "parent" directory in the present working path.
4. To return to your home directory, type **cd** and press **Return**.

HOME DIRECTORY The system administrator created a directory for you somewhere in this hierarchy. When you log in you are positioned at this directory. This directory is known as your login or HOME directory. Since this is your home directory, you should own it and all the files under it. Therefore, you have full permissions to read, write, create, change, or delete the contents of your home directory.

TERMINAL TYPE Certain UNIX commands must know what type of terminal you are using so they can interface with it correctly. You may need to contact your system administrator to find out what name UNIX uses for your terminal. The following steps show how to set your terminal type to a DEC VT100 terminal.

Other popular terminal types include:

adm3	adm5
ansi (pc)	hp
tvi925	tvi955
vt52	vt220
wyse50	wyse60

1. Type **TERM=vt100** and press **Return** to set the TERM variable to vt100 for the current login shell.

C Shell

Type **set term=vt100** and press **Return** to set the TERM variable to vt100 for the current login shell.

---

2. Type **export TERM** and press **Return** to export the TERM variable to the global shell environment. This allows other sub process commands to use the TERM variable.

C Shell

The **setenv** command performs the same function as the SV **export** command. But the csh knows to export the term variable automatically.

---

3. Type **env** and press **Return**. Notice that the TERM variable is listed as being set to vt100.

```
cj> env
LOGNAME=mylogin
MAIL=/usr/mail/mylogin
 ENV=/u1/ts/mylogin/env.ksh
HOME=/u1/ts/mylogin
TERM=vt100
PATH=:/bin:/usr/bin:/lbin:/usr/lbin:/u1/ts/mylogin/bin:.
TZ=EST0EDT
VISUAL=vi
FCEDIT=vi
SHELL=/bin/sh
```

C Shell

Type **printenv** to display environment variables.

SETTING/CHANGING YOUR PASSWORD

The **passwd** utility allows you to create or change the password associated with your user name. You can only change your password, you cannot change other users' passwords.

It is advisable to change your password the first time you log in on a multiuser system. Refer to Module 100 for a complete description of the **passwd** command.

EXECUTING COMMANDS

The command interpreter for UNIX is **ksh** (csh - BSD). The shell reads input from your terminal, interprets what you typed, and attempts to execute a program based on the input. A command is the name of a program or utility followed by options and arguments used by the program. To execute a program, the shell searches for the program based on the directory paths set in the PATH variable. The following activities will clarify command execution.

1. Display the PATH variable by typing **echo $PATH** and pressing **Return**. Notice the ":" separated fields; the ":" separates directory paths the shell searches for commands. The "." represents the "Present Working Directory."

```
cj> echo $PATH
.:/usr/lbin:/usr/local/bin:/bin:/usr/bin:/ul/ts/felps/bin
```

2. Display who is on the system by typing **who** and pressing **Return**. Notice a table of information is displayed. The leftmost column is the *user name* followed by the *port address*, and *time of login*. The shell searched the path and found the who command in the /bin directory.

3. Type **xprogx** and press **Return**. Notice that the system responds by displaying:

```
cj> xprogx
ksh: xprogx: not found
```

This tells you that your **ksh** could not locate the **xprogx** program within the directories defined in PATH.

ABORTING PROGRAM EXECUTION There are times when you decide to abort a program before it has completed execution. On UNIX you send an INTERRUPT signal to the program. To send the INTERRUPT signal, press the **Delete** key. The **csh** uses the Ctrl-C key sequence to abort an executing program. Some system administrators change the default abort or interrupt character to Ctrl-C instead of Delete or vice versa.

To display what your INTERRUPT signal is defined as, type **stty** (stty everything on BSD) and press **Return**. Look for the "intr = XX" field. The XX is the defined key. The "^?" sequence refers to the Delete key (octal 177).

COMMON COMMANDS

In this section you use the "everyday commands" of UNIX. There are commands to display, create, copy, rename, move, delete, and list files and directories. There are commands to count words, select strings, sort, print, and compare files. There are also procedures on redirecting output and using the powerful UNIX pipe commands. Begin this section at your shell prompt.

LISTING FILES UNIX provides two popular ways to list files.

1. Type **ls -la** and press **Return**. Notice that the first two lines begin with "d" and end with ".". The "d"s represent directories. The "." represents your current directory. The ".." is your parent directory.

2. Type **find . -type f -print** and press **Return**. This will display all files in you current directory and all subdirectories.

DISPLAYING FILES The following commands display the contents of ASCII files to the standard output.

1. Display the contents of a file by typing **cat /etc/passwd** and pressing **Return**. Notice the system's password file is displayed on your terminal.

2. Display /etc/passwd one page at a time by typing **pg /etc/passwd** and pressing **Return**. **pg** is useful for files that are longer than one terminal screen. Notice that the first 23 lines

are displayed and a : is displayed in the bottom left corner of your screen. To continue to the next page of text press **Return** or, for help, type **?** and press **Return**.

BSD (Berkeley)

On BSD you use the **more** command instead of **pg**.

NOTE

If the file **pg** reads less than 23 lines, then **pg** will not prompt for further input from your terminal. It basically acts like the **cat** command on short files.

3. Display the first 5 lines of the /etc/passwd file by typing **head -5 /etc/passwd** and pressing **Return**. Notice only the first 5 lines of the file are displayed on your terminal. Some systems do not support the **head** command.
4. Display the last 5 lines of the /etc/passwd file by typing **tail -5 /etc/passwd** and pressing **Return**. Notice only the last 5 lines of the file are displayed on your terminal.
5. Format the file into pages by typing **pr /etc/passwd** and pressing **Return**. Notice the file is preceded by a one-line header and has a page header every 60 lines.

CREATING FILES AND DIRECTORIES The following commands show different methods of creating files and directories.

Using Redirection The ">" symbol redirects standard output to a file.

1. Create a file by typing **cat > myfile** and pressing **Return**. Then type in the following text and press **Ctrl-D**. The ">" symbol informs your shell to *redirect* output of the **cat** command to a file, "myfile." The cat command reads from the standard input in this example.

```
cj> cat > myfile
Name:Address1:Address2:City:State:Zip
Someone:916 Hawk Street::Austin:Texas:78749
Anyone:Eagle Avenue:Suite 100:Dallas:Texas:75218
^D
```

2. Now type **cat myfile** and press **Return** to display the contents of the file.
3. Create a directory by typing **mkdir temp** and pressing **Return**. Then type **ls** and press **Return**. Notice that *temp* is now listed in your directory.

```
cj> mkdir temp
cj> ls
myfile
temp
```

4. Type **ls -l** and press **Return** to produce a long listing of your directory. Notice the *temp* line starts with a "d" and the *myfile* line starts with a "-". The "d" signifies a directory. The "-" siginifies a normal file.
5. Now set up a directory structure in your HOME directory for later use. Type **mkdir letters bin tmp** and press **Return**. This creates the "letters," "bin," and "tmp" subdirectories in your HOME directory.

Using an editor There are three popular editors on UNIX that can be used to create files. They are **ed**, **ex**, and **vi**; please refer to the module that describes each of these editors (Module 39, Module 43, and Module 151).

COPYING AND COMBINING FILES UNIX provides multiple commands to copy files. The following commands are the most commonly used.

1. Copy myfile to phone.db by typing **cp myfile phone.db** and pressing **Return**. The **cp** command does not display information if it was successful.
2. Combine two files using the cat command by typing **cat myfile phone.db > myph** and pressing **Return**. A new file called "myph" is created.
3. Display the contents of the myph file by typing **cat myph** and pressing **Return**.

RENAMING AND MOVING FILES AND DIRECTORIES The following command renames files or moves them to another directory.

1. Rename the phone.db file to phone by typing **mv phone.db phone** and pressing **Return**. Again no response if **mv** was successful.
2. Type **mv temp db** and press **Return** to rename the temp directory to "db."
3. Type **mv phone db** and press **Return** to move the phone file to the db directory.

CHANGING DIRECTORIES To move around from directory to directory you use the **cd** command.

1. Type **cd** and press **Return** to make sure you are in your HOME directory.
2. Type **pwd** and press **Return** to display your present working directory.

```
cj> pwd
/u1/ts/mylogin
```

3. Type **cd db** and press **Return** to change to the db directory.
4. Type **pwd** and press **Return**. Notice you are now in the db directory.

```
cj> pwd
/u1/ts/mylogin/db
```

5. Type **cd ../bin** and press **Return** to change to the "bin" directory that you created earlier.
6. Again type **pwd** and press **Return** to display your present working directory.

```
cj> pwd
/u1/ts/mylogin/bin
```

7. Type **cd** and press **Return** to change back to your HOME directory.

DELETING FILES AND DIRECTORIES The following commands remove files or directories.

1. Remove the files myfile and myph by typing **rm myfile myph** and pressing **Return**. The **rm** command does not respond if it was successful.
2. Type **ls -x** and press **Return** to see that myfile was removed.

```
cj> ls -x
bin     db     letters  tmp
```

3. Type **rmdir tmp** and press **Return** to remove the tmp directory.
4. Type **ls -x** and press **Return** to display the contents of your directory.

```
cj> ls -x
bin     db     letters
```

OTHER USEFUL COMMANDS There are many other commands that make life easy on the UNIX System; the more popular ones are included in the following examples.

1. Type **cd db** and press **Return** to change directories to db.
2. Type **wc phone** and press **Return** to display how many characters, words, and lines are in the file.

```
cj> wc phone
      3      7      131 phone
```

There are 3 lines, 7 words, and 131 characters in the file.

3. Select the line containing "Austin" from the file by typing **grep Austin phone** and pressing **Return**. Notice the line containing "Austin" is displayed on your terminal.
4. Sort the "phone" file by typing **sort phone** and pressing **Return**. Notice that the order of the lines are rearranged and displayed on your terminal in alphabetical order.

System V and BSD with System V extensions

5. Cut out the first field of the "phone" file by typing **cut -f1 -d: phone** and pressing **Return**. Your screen should look like the following display:

```
cj> cut -f1 -d: phone
Name
Someone
Anyone
```

---

6. Type **cd** and press **Return** to return to your HOME directory.

TEXT EDITING AND DISPLAYING

UNIX provides multiple ways to enter text into files. The most common way is to use one of the editors. The editors are **ed**, **ex**, and **vi**. The **ed** editor is the most basic editor. It is a line editor. A line editor does not display the text on the screen while you are editing. You have to display the text, then perform editing functions on a line-by-line basis.

The **ex** editor is an enhanced version of the **ed** editor. It provides much better substitution and addressing capabilities. But it is still a line editor.

The **vi** editor is a visual editor built on top of the **ex** line editor. Thus **vi** has the power of visually displaying the text you are working on while also having a powerful line editor to perform substitutions and other necessary functions.

The following steps show you how to create a file using the **vi** screen editor. It is advisable to learn the **ex** and **vi** editors because of there power and popularity.

1. Before you begin using **vi** you must have your environment set properly. First type **env** and press **Return**. Notice that a list of variables is displayed on your screen. Look for the *TERM=* variable in the list. This variable must be set before **vi** will perform correctly.

C Shell

Type **printenv** and press **Return**.

---

2. To set *TERM* to a desired terminal type, type **TERM=VT100** and press **Return**. Replace **VT100** with the type of terminal you are using.

C Shell

Type **set term=vt100** and press **Return**.

---

3. Begin editing a file by typing **vi file1** and pressing **Return**. Notice the screen is cleared and tildes (~) are placed along the left side. The filename is placed on the bottom line along with status information. Your screen looks like the following display.

```
~
~
~
~
~
~
file1 [new file]
```

4. To enter text, type **i**; this puts **vi** in *insert mode*. Now type **This is a new file created by vi.** and press **Esc**. Notice you entered a line of text and now the cursor is at the end of the line.

5. To duplicate the line, type **yy** to *yank* it to a register and type **p** to *put* it back on the screen. Notice you now have 2 lines of text that are identical.

```
This is a new file created by vi.
This is a new file created by vi.
~
~
~
~
~
file1 [new file]
```

6. To move up 1 line type **k**, to move down 1 line type **j**, to move right 1 character type **l**, and to move left 1 character type **h**.
7. To jump to the beginning of the current line type **0**. Notice the cursor moves to the beginning of the line.
8. To jump to the end of the current line type **$**. Notice the cursor moves to the end of the line. Remain in **vi** to continue the following steps.

The following steps show you how to interface with **ex** commands while in **vi**.

1. Type **:ver** and press **Return** to display what version of **vi/ex** you are using. On some systems you will need to press **Return** to redisplay the **vi** screen.
2. Write the file to disk by typing **:w** and pressing **Return**. On some systems you will need to press **Return** again to redisplay the **vi** screen.
3. To substitute a string throughout the file (global substitute) type **:%s/a new/an old/g** and press **Return**. Notice that both lines changed from having the string "a new" to having "an old." The percent sign (%) tells **ex** to search lines 1 through end-of-file (EOF). The "s" tells **ex** to perform a substitution. The "/a new/an old/" says to search for "a new" and change it to "an old." The **g** tells **ex** to change all occurrences on a line. The default is to change the first occurrence on each line. Press **Return**.
4. To escape to the shell type **:sh** and press **Return**. Notice a shell prompt appears on your terminal; you have been placed in a new subshell. You may execute any normal UNIX command. To return to **vi/ex** press **Ctrl-D** and then press **Return**.
5. To execute a UNIX command without leaving the editor type **:!cat %** and press **Return**. The % is expanded by **vi** into the current buffer name (file1). The command **cat file1** is executed and control is returned to **vi**.
6. To save the file to disk and exit **vi** type **ZZ** (capital letters).

COMMUNICATING WITH OTHER USERS

UNIX provides multiple ways to communicate with other users. There are commands to communicate interactively or by electronic mail (E-mail). The following is a brief overview of some of the more common commands used to communicate with fellow users.

WRITING TO A USER The **write** command allows you to write to a user that is currently logged on to the system. The following steps show how to "talk" to a user with **write**.

1. First you must find out who is available for you to **write**. Type **who** and press **Return**. This will display who is currently on the system. Select a user to write to and continue on to the next step.

```
cj> who
bill        tty05     Jan 11 08:41
nancy       tty07     Jan 11 08:03
mylogin     tty11     Jan 11 09:05
nasser      tty18     Jan 11 07:49
smr         tty23     Jan 11 09:11
tlp         tty09     Jan 11 07:38
```

2. Type **write bill and press Return**. Now type a message you would like to send to *bill*. When you finish writing **your message** press **Ctrl-D**.

```
cj> write bill
Hi Bill,
This is Robert. I finally got my UNIX login and am trying to
annoy all other users so they will log off the system and
response time will improve. Have a great day and LOG OFF the
system now!.
^D
```

NOTE

While you are typing your message you may see text intermixed with what you are typing. This is probably one line of a response message from the person you are writing.

RECEIVING MESSAGES FROM WRITE If someone is sending you a message from **write**, you can perform the following steps to respond.

1. First you will receive a message and a beep when someone writes to you.

```
Message from bill tty5...
```

2. Respond by typing **write bill** and pressing **Return**. From this point forward you continue your communication as if you had initiated the conversation.

SENDING MAIL You can send mail to a user by using the **mail** command.

The mail command reads in a message and sends it to a mailbox for the specific user. The following steps show how to send a simple letter to a user on your local system.

1. First you must know to whom to send the letter. To retrieve a list of all users on your local UNIX System type **nawk -F: '{print $1" "$5}' /etc/passwd | sort | more** and press **Return**.
2. Select a user name and type **mail** *nancy* and press **Return**. Now you may type the letter you wish to send to *nancy*. After you finish the letter, press **Ctrl-D** to send the letter to *nancy*. Replace *nancy* with the name of a user on your system.

RECEIVING MAIL To receive mail, a user must first send mail to your user name. If no one likes you and you never receive mail you can always send mail to yourself. This is very common practice; it provides an on-line reminder service. The following steps show how to read any mail that may have been sent.

1. Type **mail** and press **Return**. Notice the message you requested your system administrator to send you is displayed. If there's no mail you probably have a very busy system administrator.
2. The **mail** utility has a help feature that provides some insight to using it more effectively. Type **?** or **help** and press **Return**. Notice a screen of information on the mail utility is displayed.

NOTE

There are two major versions of the mail utility. One is UNIX System V and the other is Berkeley based. The Berkeley version is supplied with System V as **mailx**, although it is possible that it has been moved to the **mail** name. This book discusses the Berkeley /usr/ucb/Mail version and the System V /bin/mailx version of mail and refers to it as **mail/mailx**.

LOGGING OUT OF THE SYSTEM

To log out of UNIX you must exit the shell. There are multiple ways of exiting the shell; the most common is pressing **Ctrl-D**. You may also type **exit** and press **Return** or, on some systems, you may type **logout** and press **Return**.

Turn to Module 3 to continue the learning sequence.

Module 3
SYSTEM OVERVIEW

DESCRIPTION

This module provides a general overview of UNIX and the UNIX User Utilities described in this book. It specifically covers the following information:

- General overview
- The UNIX Kernel (the operating system)
- Multitasking
- Multiuser
- Portability
- Job control
- File System
  - Directories
  - Ordinary Files
  - Special Files
    - Fifo (pipe)
    - Device Drivers
- The UNIX shell
- System Documentation

GENERAL OVERVIEW

The UNIX operating system has rapidly grown in popularity in recent years. Every major computer manufacturer offers a version of UNIX for their hardware, a claim only UNIX can make. It has been ported to over 150 different types of computers, ranging from micros to supercomputers.

The philosophy of UNIX is to modularize problems and build on existing software solutions. Each program should be designed to solve one problem, and complex problems should be solved by combining multiple programs. This basic idea of UNIX is often referred to as the "building block" approach. Elegantly designed as a simple yet extremely powerful general purpose operating system, UNIX has become a standard in the computer industry.

The UNIX System has introduced many new features to the computer world. Because of its rapid growth in popularity it has become the center of much praise and controversy. It is often referred to as a programmer's "dream" and a user's "nightmare." UNIX is a tremendous solution to many problems, but there are definite weaknesses that are being rectified as new releases are developed. The user interfaces that were once foreign to UNIX are now a reality. There are many software packages available today for UNIX.

THE UNIX KERNEL

The *UNIX kernel* is the operating system program of UNIX. It is a C and assembly language program that provides the low-level functions of the "UNIX environment." It controls the computer hardware and provides an interface for all other UNIX programs to use the hardware. This concept allows the higher level programs to be hardware independent. When a program needs service to a hardware device, it calls on the kernel to perform the function.

The kernel provides the user the ability to interface with the computer hardware and peripherals. When you log in to UNIX you communicate with the kernel via a shell program. Seldom does a user interact directly with the kernel.

MULTITASKING

Multitasking refers to an operating system that executes *multiple tasks* simultaneously. UNIX refers to a task as a process. UNIX can be configured to allow the user to run from one to as many processes as the operating system can handle. The standard is usually a maximum of 25 processes or tasks per user login. The theory of a multitasking system is to approach 100 percent in computer resource utilization while increasing user productivity. Of course, at some point, one or more resources are over used and productivity begins to fall instead of continuing to rise.

A user can run several commands in *background* while executing another command in *foreground*. Background refers to commands that are detached from your terminal. Foreground is the execution of a command attached to your terminal. You must wait for the current foreground command to complete before you can execute another foreground command. Once a command is entered, the system breaks the command down into separate processes if possible. Each process equates to one program or utility. Internal ksh commands do not always generate (spawn) a new process.

MULTIUSER

Multiuser refers to an operating system that allows *multiple users* to use the system simultaneously. UNIX currently runs on systems that support from one to over 2000 users. The theory of a multiuser system is to approach 100 percent in computer resource utilization while reducing the cost per user. This is done by having multiple users share system resources. A single user cannot use the printer, disk, memory, or cpu 100 percent of the time. But multiple users can increase use of these devices and resources by having an operating system that manages the resources for them. This reduces the cost per user of the computer as more users share the system.

Computers with enough resources to share among multiple users are prime targets for multiuser operating systems. The main requirements are enough RAM memory, disk storage, terminal ports, and cpu power to handle the operating system and each user's computing needs. Even "old" micros have this capacity — the popular Intel 80286 or the Motorola 68010 can run 1 to 8 users with relatively good response to each user. The key to good response on UNIX is not just the cpu speed, but a fast access time on your disk drives. The faster your disk drives (low access time) are, the better UNIX responds. It is also very important to have fast I/O to disk and terminal ports.

PORTABILITY

One of the most sought after features UNIX possesses is portability. Portability is the ability to rewrite the operating system for a different vendor's hardware without a major rewrite effort. UNIX is highly portable and has been ported by many hardware vendors to their computer

hardware. In fact, every computer vendor has a UNIX or a UNIX-like system ported to their hardware — a milestone that no other operating system can claim.

There are two main reasons UNIX is a prime system for porting from hardware to hardware. The first is that the kernel provides an interface between the hardware and most of the nonkernel UNIX software. When UNIX is ported to another hardware platform, only the kernel requires major modifications. Since most of the UNIX software interfaces with the kernel and not the hardware, the software is hardware independent and does not require much, if any, modification when it is ported to a new hardware platform.

The second reason UNIX is easy to port is that the majority of the UNIX operating system is written in the C programming language. Most operating systems are written in the hardware vendor's proprietary assembly language. This makes it very difficult, if not impossible, to port their operating systems to a different vendor's computer.

Benefits Portability allows the customer to choose the vendor and not be locked into that vendor's hardware and software environment. It also provides the user with a broad base of application software that can be used on hardware from different application vendors.

The porting of an operating system creates a standard software environment across a broad range of computers, ranging in size from micros to supercomputers. This cuts training time dramatically and increases overall productivity. It also reduces support costs and redevelopment costs.

PORTABLE APPLICATION SOFTWARE The UNIX System provides a standard application development environment that allows for easy porting of application software. The C programming language is a big asset to the porting of software between different UNIX machines.

The ability to port software to many systems easily provides application software vendors with a much larger market without the headaches of multiple proprietary environments. Customer training requirements are reduced and the training of internal personnel is minimized. The number of support people needed is not increased because of different knowledge base requirements, but because more software is being sold. Development can spend more time on enhancements instead of ports to unknown, proprietary architectures. Once a customer has bought a product on one UNIX machine, the customer can purchase a different vendor's UNIX machine and be able to use the same applications without having to retrain personnel. As you can understand, portability is a very important issue for many computer users.

JOB CONTROL

Job control on UNIX is the ability to control which job is executed in foreground, background, or is suspended.

- *Foreground* execution is considered normal interactive command execution. The command is entered from the keyboard. The shell waits until the command completes execution, then the shell prompts for another command to execute.
- *Background* execution is considered batch processing. A command is entered and detached from the terminal. The shell can continue interactive command processing.
- *Suspended* jobs (processes) are commands that have been placed in a suspended state. They are not being executed in background or foreground.

Job control allows you to suspend a foreground or background job. A suspended job can be moved to background or foreground. Background jobs can be brought to the foreground and foreground jobs can be placed in the background.

Using job control can increase the productivity of a user by allowing multiple tasks to be juggled back and forth between background, foreground, or a suspended state. For example, a user can edit a source file while executing the compiled program, thus watching the functionality of the program and making changes while suspending the program.

FILE SYSTEM

UNIX uses a hierarchical structure to store and maintain files. This structure is referred to as a tree structure from its resemblance of an upside down tree. The UNIX file system starts with one master directory called "root." Since the tree is upside down, the "root" directory is at the top. The root directory, symbolized by a single "/", has multiple directories under it.

This tree structure allows maximum flexibility for storing information to resemble its logical or real life structure. Multiple users may be grouped by corporate department. As an individual user you may group your data by project or subject. The options are almost limitless and the result is a very flexible and productive environment.

The file system is used by the UNIX operating system to store and control system and user files, directories, and special files. It allows a logical and simple method to maintain large quantities of files with little effort.

The file system operations are performed internally by the UNIX System but there are many utilities that can be used to manage and manipulate the information stored within a file system.

The UNIX operating system has three concepts of files within the file system, the *directory*, *ordinary file*, and *special file*. When using the term *file* we are usually referring to an ordinary file.

Characteristics The following list describes the characteristics of the UNIX file system.

Hierarchical structure — Allows grouping of related information and efficient manipulation of these groups.

Dynamic file growth — Files grow as needed, no advanced decisions have to be made about file sizes. Disk space is not wasted since only the amount required to store the current contents of the file is used.

Structureless files — There is no internal structure imposed on the contents of a file. Any structure desired by the user can be used.

Security — Unauthorized users can be restricted from accessing a file.

Device independence — Input and Output from a device is processed as if it were in a file. Therefore, programs that process data in a file can also process data to and from a device.

Limitations File sizes cannot exceed 4 gigabytes (2^32). As files increase over 10,000 bytes in size, the access time required to retrieve the data increases. This is caused by indirection to locate the block of data.

As the number of entries in a directory increases, the access time to view or manipulate the directory increases. In-core (file system information stored in RAM) file system information may be lost if the system crashes.

DIRECTORIES A directory is a storage area where any combination of files and subdirectories can be managed. You can think of a directory as a filing cabinet drawer, used to store related file folders. Each directory has a name and each file within the directory has a filename.

Directories are used to store multiple files according to the needs of you, the user. Some people organize their directories according to subjects. Others may organize them based on projects or time schedules. The point is to utilize the freedom provided by the file system to organize them as you see necessary.

The UNIX System is distributed with an established set of directories. There are many system directories and possibly locally created system directories. Check the vendor's Release Notes, Install Procedures, or System Administrator's Guide for a full list or ask your system administrator.

The / (root) and /usr file system must be mounted for the system to be fully functional. The root file system contains the essential files and utilities. The /usr file system contains more general files and utilities. On smaller systems the /usr directory is not a file system but is part of the / file system.

Limitations As with a physical file cabinet drawer you can only put a specific number of file folders in the drawer. A directory has a physical limit of files in the thousands.

NOTE

It is advisable to limit the number of files in a directory to under 100. Some systems can actually handle up to 500 files in one directory but 100 is still a good limit. There are two major reasons for this limit. The first is performance. If more than 100 files exist in a directory, the UNIX System encounters high overhead to perform various file system functions in that directory. The second reason is user sanity, referred to by some people as "organization." If you try to keep too many files in one directory, you begin to lose track of where you stored your data.

The directory name can be a maximum of 256 characters, consisting of any character, including a white space. Some characters may have to be escaped or quoted to perform this trickery, but it is possible, though not often desired. Some systems allow up to 1024 characters in the filename.

ORDINARY FILES Ordinary files are used to store data. A file is like a folder in a filing cabinet; many different types of information may be stored in the file. Files may be ASCII text, compiled programs, or free formatted application controlled data. Files may grow as needed without intervention from the user. This makes it easy for the user and saves space on the physical storage device.

Internal Structure An ordinary file has no internal structure defined. The data may be stored in any format desired. The UNIX System views files as a stream of bytes, it reads and writes them byte-by-byte or block-by-block, not interpreting the data.

UNIX identifies files by a unique number called an inode number. Inode numbers are maintained in a directory file along with the related filename. If you refer to a file by its filename, the system uses the related inode number to retrieve the physical data.

Each file has one inode number that points to that file's index node, or inode for short. A file has one and only one inode, even though it may have multiple filenames. The filenames are all linked to the same inode by the directory entries. An inode contains the following information:

owner	(user ID)
type	(directory, ordinary, block special, character special, or pipe)
access permissions	(user, group, others)
access times	(last modification, last access, last inode modification)

number of links	(number of filenames stored in directories)
size in bytes	
disk addresses	(Multiple entries point to the data of which - some entries point to data blocks - other entries point to a single indirect block - other entries point to a double indirect block - other entries point to a triple indirect block)

Limitations The current limit on a file size is over 4 gigabytes, based on 1024-byte file system blocks, usually more space than is available on the entire system's disk drives. The name of a file can only be 256 characters.

Problems On a multiuser system with thousands of files, there is the possibility that the filename has been used previously by another user. This does not pose a large problem, because most users have their own private directories, but it must be considered in publicly used directories.

SPECIAL FILES Special UNIX files are pieces of computer code that interface with the UNIX kernel. There are two types of special files: device drivers and fifos. Both are interfaces within the UNIX kernel, a part of the I/O subsystem.

Device drivers are used to communicate with and control peripheral devices attached to your UNIX System, such as printers, terminals, disk drives, or tape drives. Device drivers may be block or character special, referring to how the driver handles I/O to and from a device.

Fifo (Pipes) are used to join two or more UNIX processes together allowing the streaming of data. Streaming data means that data flows from one program into the next without being stored on disk. Fifo special (first-in first-out), or named pipes, allow unrelated UNIX programs to exchange information.

Device Drivers Device drivers are block or character special files. They are low-level routines that reside in the system kernel where they interface to peripheral devices, such as terminals, printers, disk and tape drives, connected to the computer system. They perform I/O tasks to and from a specific type of device. UNIX device drivers are interfaces that make devices appear as ordinary files when performing I/O functions.

Block-type drivers read and write in blocks of data, resembling random access storage. A block is usually 512 or 1024 bytes, depending on your system's BSIZE constant stored in /usr/include /sys/param.h.

Character-type drivers read and write a single byte of data, often referred to as raw devices. Most of the device drivers on a UNIX System are character special.

Pipes A pipe is a UNIX file that takes the output of one process and provides it as input to the next process. A more internal view of a pipe would be that data is transferred between processes in a first-in first-out sequence. The pipe also synchronizes the execution of the surrounding processes. The first process must execute and begin output before the next process can begin its execution.

The most common use of pipes is to connect multiple UNIX programs together to form a pipeline. A pipeline feeds information from one program to the next.

When executing multiple UNIX commands, pipes may be used between the commands to create what is referred to as a pipeline command. Likewise, if data is to start in an original state at the source and be manipulated by commands along the way to its destination, then pipes must be used

to connect the commands together so that the data flows to the destination. For convenience, the entire command is called a pipeline.

Since the commands are changing the data along the way, they are referred to as filters. A command must be capable of being a filter for it to be used in a pipe. A filter can read from standard input and write to standard output. The pipe passes data unchanged while a filter manipulates the data as it is passed down the pipeline.

Enter the following command at your shell prompt and press **Return**:

```
cj> grep :0: /etc/passwd | nawk -F: '{ print $1 " "$5}' | sed 's/:/<TAB>/g'
root          Admin. - super user
sync          File system sync
dgn           Diagnostics
```

The output is the first and fifth fields of lines from the password file that contained the :0: string, which should be all logins with root user ID permissions.

The same results could have been accomplished using temporary files, but this is very inefficient and clumsy.

```
cj> grep :0: /etc/passwd > TMP1
cj> nawk -F: '{ print $1 " " $5 }' TMP1 > TMP2
cj> sed 's/:/<TAB>/g' TMP2
root          Admin. - super user
sync          File system sync
dgn           Diagnostics
cj> rm TMP1 TMP2
```

THE UNIX SHELL

The shell (ksh) is a very powerful and dynamic UNIX utility that functions as the UNIX *command interpreter*. It is the primary user interface to the operating system (kernel). The shell is a command language and a programming language. As a command language it can be used to communicate interactively with the kernel. As a programming language users can write shell scripts to solve simple to complex problems.

The shell is probably the most sophisticated UNIX utility. It can read from your terminal or from a file. Thus is can be interactively programmed or programs can be stored in files.

The following is a typical sequence of events when a user interacts with the shell as the primary user interface:

1. User logs in to the system.
2. The shell prompts for input (a command).
3. User types in command and presses Return.
4. The shell searches for program; if found, executes it.
5. Program runs; user may interact with program.
6. Program exits; user may have to exit the program.
7. The shell prompts for next command; return to number 4 above.
8. User types exit, logout, or Ctrl-D to exit shell.
9. The shell logs the user off the system.

Modules 71 and 23 include detailed looks at the shell as a command language. Module 108 is a detailed look at the shell as a programming language.

SYSTEM DOCUMENTATION

There are many different documents produced by AT&T and other UNIX developers. The primary documents for each UNIX System are the Reference Manuals and Guides. These documents provide you with specific details on the UNIX System you are using. One other document that most users ignore but that is extremely important is the Release Notes for UNIX and all related software. Release Notes provide insight to changes, quirks, problems, and compatibility issues relating to the new release of the software. It will be to your benefit to locate a copy and discover what your release of UNIX holds in store for you.

The *User's Reference Manual* and the *User's Guide* describe the basic utilities and software of the UNIX System. The Reference Manual describes each utility that is available on the UNIX System. The Guide provides an overview and tutorials on the more common and complex UNIX utilities.

Illustrated UNIX addresses both of these areas to provide a tutorial and a reference manual in one book. It demystifies the guru language of the standard reference manuals and organizes the tutorials into a useful sequence that can later be easily referenced.

Turn to Module 71 to continue the learning sequence.

Module 4
alias/unalias (csh, ksh)

DESCRIPTION

The internal **alias** command provides a shorthand for one or more commands. The alias command is supported by the csh and the ksh. There are three types of aliases in the Korn shell: exported, preset, and tracked. An exported alias is passed to subshells when called by name. The preset aliases are set by the Korn shell. The tracked aliases are full pathnames for commands.

Aliases can be used to redefine the shell's built-in commands but not the *keywords*.

Aliases can be created, exported, and listed using the **alias** command. Aliases can be removed using the **unalias** command. Exported aliases are passed to subshells. Therefore, if you have an alias defined in your login shell and you execute a shell script, the shell script is able to access the aliases.

Aliases are established as they are read, not during execution. For an alias to work, it must be created before the command that uses the alias is executed. Basically, you have to define your aliases before you can reference them.

The internal **unalias** command is used to remove alias definitions. For example, if you want to do away with your *l* alias, you enter *unalias l* and press Return.

Aliases can be created, exported, and listed using the **alias** command. Aliases can be removed using the **unalias** command.

COMMAND FORMAT Following is the general format of the *alias* command.

```
alias [ -tx ] [ name=string ... ]
alias name
unalias name
```

If no options or arguments are given, then all existing aliases are listed.

C Shell (csh)

```
alias name string
unalias name
```

Options The following options may be used to control how **alias** functions.

–t Allows you to set and list tracked aliases. A tracked alias has a full pathname for the command used in the *string*. This provides less overhead in search and execute time since the location of the command is already known. If your PATH variable is reset, the string of a tracked alias is reset to NULL. If no arguments are provided on the command line, all tracked commands are listed — not really an alias, but it is how the Korn shell handles tracking commands.

-x Allows you to set or list exported aliases. An exported alias functions like an exported variable; it may be passed to subprocesses. If no arguments are provided on the command line, all exported aliases are listed.

C Shell

The **csh** alias does not support any options.

Arguments The following list describes the arguments that may be passed to the **alias** command.

name=string Defines an alias *name* that runs the command *string*.

name The name of the alias. The name becomes a new command that is an alias for the command specified in the *string* part of the alias. For example,

alias lx="ls -x"

sets the name **lx** to the command **ls -x** specified in the string. Now you can use **lx** as a command.

If only a *name* is provided, then the alias *string* for that alias is displayed if it exists.

= The delimiter between the *name* and command *string*.

string The command you wish to have executed when you use *name* as a command. If any spaces or tabs are contained in the string, then you must quote the entire *string*. For example,

alias dir="ls -la | pg"

sets **dir** to be an alias for the command **ls -la | pg**. So when you type dir and press Return, you actually execute the ls -la | pg command, the *string* part of the alias.

If a space follows the string of an alias command, then the following commands are checked for being an alias.

If no *names* or *strings* are given, then the appropriate aliases are listed with their corresponding string.

If there are spaces in the *string* side of the alias, you must enclose it in quotes.

C Shell (csh)

The C shell uses a slightly different syntax to set up an alias. Instead of using an equal sign (=), you simply put a space between the *name* and the command. The *string* does not have to be enclosed in quotes if it contains spaces.

The C shell alias also allows you to use history substitution which is replaced by command line arguments. For example,

```
alias la 'echo \!*'
```

would expand la * into echo file1 . . ., which would be all the files in the current directory.

FURTHER DISCUSSION If you alias a command that already exists as a true command, you can still refer to it by enclosing it in single quotes. For example, if you set the following alias

```
alias ls='ls -x'
```

you can still run the normal *ls* by typing

```
'ls'
```

or

```
/bin/ls
```

It is considered best not to rename an existing command using **alias** unless you always use the command the way you are setting up the alias. For example, if you always use **ls -x**, you may want to create an alias, **ls="ls -x"**. To keep from interfering with the **ls** command, you might consider using **l** or **lx** for the alias name.

If you wish to have a command aliased each time you log in to the system, you must place the alias command line in your .profile or in your ENV (.kshrc) file. The ENV file is executed each time a **ksh** is started.

You might find the following list of aliases useful.

```
alias -x lf="ls -Fx"
alias -x l="ls -l"
alias -x ud="cd .."
alias -x bd="cd -"
alias -x dir="ls -l"
alias -x dir/w="ls -x"
alias -x copy="cp "
alias -x more="pg -n "
alias -x type="cat "          # although type is an internal command
alias -x prt="lp -dlaser"     # alias to whence using print is not advisable
```

Exported aliases Exported aliases are like exported variables. They are passed to subshells. Aliases that have not been exported are not passed to subshells and, therefore, are not known by such subshells. The following aliases are automatically exported by the shell.

```
echo='print -'
false='let 0'
functions='typeset -f'
hash='alias -t'
history='fc -l'
integer='typeset -i'
nohup='nohup '
pwd='print - $PWD'
r='fc -e -'
true=':'
type='whence -v'
```

Tracked aliases Tracked aliases are not true aliases. A tracked alias is the actual command name and the full pathname used to access an existing command. It is called tracked alias because the path is aliased for the command name and the new alias tracks directly to the command. This keeps the shell from having to search the entire PATH for a command, thus providing faster command execution.

You can turn command tracking on by using the **set** command. Type

```
cj> set -o trackall
```

to inform the shell to track all commands issued to it. For example, the following sequence of commands illustrates the automatic tracking of commands.

```
cj> set -o trackall
cj> alias ls                    # list alias of ls command
ls alias not found              # no alias exists for ls
cj> ls                          # execute the ls command
...
cj> alias ls                    # list alias of ls command
ls=/bin/ls                      # tracked alias has been set to full path
```

You can track individual commands by using the -t option. For example,

```
cj> alias -t ls
```

would cause the same results as the above steps. But the shell would not automatically set every command you issue to a tracked alias.

RETURN CODES Alias returns a zero (true) unless a *name* is given that has not been set to a string.

APPLICATIONS

The **alias** command is used to create shorthand versions of longer commands. It is very useful for hard-to-remember commands and for those commands you use repeatedly throughout the day. Remember to place your aliases in your .profile or your ENV file (.kshrc) so they are set each time you log in.

The **unalias** command is used to remove previously set aliases.

TYPICAL OPERATION

In this activity you use the **alias** command to create various aliases. Begin at the shell prompt.

1. Type **alias "l= -F"** and press **Return**.

C Shell

Type **alias l "ls -F"** and press **Return**.

---

2. Now try your new command by typing **l** and pressing **Return**. Notice the output of your **l** command is a multicolumn list with the executable (*) and directory (/) files being flagged.
3. List all aliases you have set by typing **alias** and pressing **Return**.

In this activity you use the **unalias** command to remove the previously defined **l** alias command. Begin at the shell prompt.

1. Remove the **l** alias by typing **unalias l** and pressing **Return**.
2. Type **alias l** and press **Return** to list out your **l** alias. Notice the **l** alias is no longer defined.
3. Turn to Module 64 to continue the learning sequence.

Module 5
apropos (BSD)

DESCRIPTION

The external **apropos** command searches for keywords in the header lines of all UNIX Reference Manual entries. There is one manual entry for each UNIX command distributed with your system. These entries are often referred to as "man pages." The header line for each command entry is a brief one-line description of the command.

If you need to use a command but do not know its name, you can use **apropos** to search for the command by specifying keywords. You specify keywords on the **apropos** command line. Each keyword is searched for separately in the header lines of all manual entries. If a keyword is located in the title (header line) of a man page, the command's title is displayed. The search ignores the case of letters.

A match occurs even if the keyword is a part of a word in the title. For example, if you specify

```
apropos command
```

all titles with the word "command" and "commands" would be displayed.

COMMAND FORMAT Following is the general format of the **apropos** command.

```
apropos keyword [ ... ]
```

Arguments The following list describes the arguments that may be passed to the **apropos** command.

keyword A word you want to search for in the title of all the manual sections.

FURTHER DISCUSSION The **apropos** command is the same as the **man -k** keyword command. If **apropos** displays a line for a keyword that begins with *name(section)* ..., you can display the entire manual page by entering the command:

```
man section name
```

To display the title line of each manual page that contains the keyword "editor" you specify a **apropos** command like:

```
apropos editor
```

which should return the following output:

```
a.out (5)        - assembler and link editor output format
ed, red (1)      - basic line editor
ex, edit (1)     - line editor
ld (1)           - link editor
sed (1)          - stream text editor
vi, view (1)     - visual display editor based on ex(1)
```

The output from your system may vary depending on which utilities have been implemented and installed on your particular system.

DIAGNOSTICS AND BUGS If **apropos** fails to work, it is possible the */usr/lib/whatis* file has not been created. The system administrator creates this file by changing directories to /usr/lib and executing **mkwhatis** or **makewhatis**, depending on your system's command name.

RELATED COMMANDS Refer to the **man** and **whatis** commands described in modules 87 and 155.

RELATED FILES The **apropos** command looks in a file containing the titles for all of the manual documents, usually the */usr/lib/whatis*.

APPLICATIONS

The **apropos** command provides a way for you to search for a command to perform a certain function based on keywords. For example, if you needed to display the contents of a file, you could use apropos and specify several keywords that referred to displaying contents of files. For example, you might try the command,

```
apropos print display show type file
```

The trick to making apropos work is being flexible in your choice of keywords. Flexible means you should try to remember as many synonyms for a word as you can.

There are trade-offs in how you use keywords. If you use too many or too broad of a word, such as *file*, you may get pages of output. On the other hand, if you use a very selective set of keywords, you may get no output. The basic advice is to try a few different keywords to get the general idea of how the titles of the man pages are written. Unfortunately, experience is still the best teacher, especially when UNIX man pages are involved.

TYPICAL OPERATION

In this activity you use the **apropos** command to locate commands related to *text*. Begin at the shell prompt.

1. Type **apropos text** and press **Return**. Depending on your system, the output should resemble the following:

```
cj> apropos text
nawk(1)                 - text pattern scanning and processing language
csplit(1)               - context split
ed, red (1)             - text editor
edit(1)                 - text editor (variant of ex for casual users)
end, etext, edata(3C)   - last locations in program
ex(1)                   - text editor
fspec(4)                - format specification in text files
lex(1)                  - generate programs for lexical analysis of text
neqn(1)                 - format mathematical text for nroff
nroff(1)                - format text
plock(2)                - lock process, text, or data in memory
sed(1)                  - stream text editor
```

2. Turn to Module 65 to continue the learning sequence.

Three Words <*\\ Sophesticated data type*//*/>

{"Novell Netware",
"Predator",
"Angular"
};

Module 6
awk/nawk

→ 888
MANGALIC
1000 → 100
OOP → D<*\\ 500 //*/>

DESCRIPTION

The external **awk** program is an interpretive programming language. Interpretive means **awk** executes the program as it reads the code, unlike the C programming language which must be compiled before it can be executed. **Awk** is somewhat of a cross between the **egrep** command and the C programming language. Like **egrep** it searches for a regular expression in the input. Like C it can be programmed to perform almost any data manipulation processing.

INT-ER-PRE-TI-VE.

NOTE

The new version of **awk** is named **nawk** (new awk). Some vendors provide only the old version of awk, some provide nawk under the name of awk, while others provide awk and nawk. System V Release 4.0 provides awk and nawk. It is advisable to use nawk unless of course you have to consider portability to systems that do not support nawk. This module describes the nawk program.

For "TC" 14th OCTOBER 2015

Nawk lends itself well to generating reports, transforming data, retrieving information, and validating data. It is designed to handle data in rows and columns, but it is also useful in locating specific data within free formatted data streams. Its features include:

- Structures, operators, and syntax resembling the C programming language
- Regular expression pattern comparison
- Multisource input. It can read from files, pipes, and the keyboard in the same program. It can be interactive.
- Multidestination output. It can write to multiple files, pipes, and the terminal.
- Formatted output using the C printf function
- Automatically declares typeless variables **(a variable** may be *string* or *number*)
- Automatically parses input lines into **fields (columns)** and records (lines)

B.A.S.I.C.

COMMAND FORMAT Following is the **general** format of the **nawk** command.

```
nawk [ -Fs ] program [ VAR=VAL ] [ file_list ] [ - ]
nawk [ -Fs ] -f prog_file [ VAR=VAL ] [ file_list ] [ - ]
```

Options The following list describes the options and their arguments that may be used to control how **nawk** functions.

-F*s* Set field separator to regular expression *s*; default is white space (spaces and/or tabs).

WC / WK <*\\ event, interrupt, signal, timer, trigger //*/>

-f prog_file The -f informs **nawk** that the next argument is the name of the file containing the nawk commands. The file *prog_file* contains the nawk program (commands). This allows you to write large nawk programs without exceeding the shell's or nawk's input buffer limit on the command line. The input buffer

K → Kassena

N-agma SVille

on most systems is between 512 and 10240 bytes. To find out your system's limit, type grep "ARG_MAX" /usr/include/limits.h and press Return. On BSD systems, type egrep "NCARGS" /usr/include/sys/param.h and press Return.

Arguments The following list describes the arguments that may be passed to the **nawk** command.

'program' A single argument containing the **nawk** program (commands). The argument should be enclosed in single quotes to keep the shell from interpreting it.

VAR=VAL Command line variables. These variables are passed to the **nawk** *program* for internal use.

file_list One or more files containing data to be scanned and processed by **nawk**.

– The standard input is read as input. This may be intermixed with files in the *file_list*.

If no files are specified for input, **nawk** reads from the standard input.

DATA FILE The examples throughout this module assume you have a file named *fruits* in your HOME directory. The file should contain the following data.

```
FRUIT     STATE       BUSHELS     PRICE    STATUS
Apples    Washington  6,700,000   10.59
Oranges   Florida     5,900,000   11.69
Peaches   Texas       3,600,000   13.79    Incomplete
Pears     California  3,100,000   12.89
Raisins   California  2,300,000   15.00
Cherries  Missouri    2,100,000   17.49
Pineapple Hawaii      3,900,000   12.99
Coconuts  Hawaii      2,600,000   14.39    Incomplete
```

If an example is given for syntax reasons only, the filename *infile* is used.

PROGRAMS The **nawk** command interprets a set of commands known as the **nawk** program. The program can be specified as an argument on the shell command line or stored in a file. In either case the program consists of one or more *pattern-action* statements. The general form of the pattern-action statement is:

```
pattern { action }
```

For example,

```
nawk 'NF < 5 { print $1 }' fruits
```

prints the first field of a line if less than five fields exist on the line. Fields are space or tab separated words. The built-in variable NF contains the number of fields in the current line.

Nawk reads each line of input automatically. If the *pattern* matches the input line (record) in part or whole, then the following *action* is performed. If no *pattern* is specified, then the *action* is performed for every input record. The action may consist of multiple programming statements discussed later. The action must be enclosed by '{}' for **nawk** to function properly. The general format is:

```
{ action }
```

{grapple}

For example,

```
nawk '{ print }' fruits
```

prints every line of the file *infile*. Thus you wrote **a very simple** version of the **cat** command. This is the default pattern-action statement of **nawk**.

If no *action* is specified, each matching input record is displayed on the standard output. The general format is:

holy/holi

```
pattern
```

[?????] H → ola Append → holx
ole → holy
oli
olo
olu

For example,

```
nawk 'NF == 4' fruits
```

prints each line consisting of four fields.

WRITING PROGRAMS It is very common to write small **nawk** programs in-line; that is, on the command line or even spread over a couple of lines. For example,

```
nawk 'BEGIN { print "Annual Fruit Report" }
      { print }
      END { print "More fruit to Grow" }' fruits
```

prints the banner line *Annual Fruit Report*, every line of the *fruits* file, and a trailer line *More fruit to Grow*. This line can be typed from your terminal or placed in a shell script.

NOTE

In-line **nawk** programs are limited in length to the command line buffer of your system. This buffer varies; some systems are set to 512 characters of information while others are set to 10240 characters. The general rule of thumb for the length of in-line programs is that they should never exceed 2000 characters (approximately one screenful of characters).

↳ Rawk

The same **nawk** program written in a file format would be invoked by the command line:

```
nawk -f fruitprog fruits
```

The *fruitprog* file would contain the following lines.

```
BEGIN { print "Annual Fruit Report" }
{ print }
END { print "More fruit to Grow" }
```

Notice the single quotes are removed; the **nawk** command itself is not needed for the input files.

PATTERNS Patterns are used to select lines from the input. If a pattern matches a string on an input line, the associated action is performed. It is possible to write an **nawk** program that does not have a pattern, such as

```
nawk '{ print $1 }' fruits
```

The default pattern selects all lines from the input.

NOTE

The BEGIN and END patterns both require an action.

The BEGIN Pattern The BEGIN pattern does not match any input. Instead, the action associated with the BEGIN pattern is executed before any input is read by **nawk**. This allows for

initialization of variables, the printing of headers, and other coding that needs to be done before the first line of input is read by **nawk**. The general format is:

```
nawk 'BEGIN { initialization code; headers; etc. }' infile
```

The following example prints a header line.

```
nawk 'BEGIN { print "Fruit Shipped From Primary State" }' fruits
```

The END pattern The END pattern does not match any input. Instead, the action associated with the END pattern is executed after all of the input has been read by **nawk**. The general format is:

```
nawk 'END { wrap up code }' infile
```

The following example prints the total number of lines read by **nawk**.

```
nawk 'END { print NR }' infile
```

The built-in variable NR contains the Number of Records read.

Regular Expression Patterns The following table contains each regular expression and the task that it performs when used inside a pattern. Regular Expressions are often referred to as REs in UNIX terminology. Thus we use the RE notation for uniformity and briefness.

Metacharacters Metacharacters are the special characters used in regular expression patterns that have special meanings. Metacharacters are often referred to as *special* or *magic* characters.

Regular expression patterns must be enclosed in slashes. For example, to match any records (lines) containing the string *cpu* you would use /cpu/ as the pattern. The string inside the slashes may be any valid combination of the following list.

Special RE	*Description*
c	Matches the character *c* if *c* is not a special regular expression character.
\	Escapes the meaning of a metacharacter.
^	Matches the beginning of the line.
$	Matches the end of the line.
.	Matches any single character other than the new-line.
[*class*]	A character class. Matches any one character in the class.
[*c1-c2*]	Matches any one of the ASCII characters in the range defined within the brackets.
[^*class*]	Does NOT match any of the ASCII characters listed within the brackets. Ranges may be specified.
|	Alternation of regular expressions. Matches either one or the other of the regular expressions provided.
(!)	Concatenation operation. Normally the parentheses are omitted. Allows for control of precedence in the interpretation of the regular expressions specified.
RE*	Matches zero or more occurrences of the preceding regular expression.
RE+	Matches one or more occurrences of the preceding regular expression.
RE?	Matches zero or one occurrence of the preceding regular expression.
//	The null RE refers to the last RE defined.

The following example matches any line containing the word *Apples*.

```
nawk '/Apples/' fruits
```

Escape Sequences The following sequences of characters are referred to as *escape sequences*. Escape sequences specify a notation for characters that are not easily produced from your keyboard and/or cause problems when displayed to your screen. The **nawk** and **echo** commands use escape sequences. Although only **nawk** uses the escape sequences in regular expressions.

Sequence	*Description*
\b	Matches a backspace.
\f	Matches a form feed.
\n	Matches a new-line.
\r	Matches a carriage return.
\t	Matches a tab.
\0ddd	Matches the ASCII character for the given octal ASCII code.
\c	Matches the literal character *c*. Use \\ to generate a \.

Expression Patterns Expression patterns perform number or string comparisons. **Nawk** provides eight comparison operators. Six of these operators are relational operators used to perform number or string comparison. The other two are string operators used to perform string comparisons.

String Operators may be used to compare a regular expression pattern to a specific field or the entire input line. The following two operators are supported:

Operator	*Function*
/RE/	Matches if the current line contains the specified regular expression. Same as the $0 ~ */RE/* command.
expr ~ RE	Matches if the string value of *expr* contains the regular expression *RE*. For example, `nawk '$1 ~ /[Aa]pples/' fruits` matches lines where field 1 contains "Apples" or "apples."
expr !~ RE	Matches if the string value of *expr* does not contain the regular expression *expr*. For example, `nawk '$3 !~ /[Aa]pples/' fruits` matches lines where field 1 does not contain the word "Apples" or "apples."

Relational Operators Relational operators compare strings and numbers. The following six operators are used to compare values.

Operator	*Function*
lv > rv	True if *lv* is greater than *rv*. For example, `nawk 'NF > 3' fruits` prints each line containing more than three fields.

lv >= *rv*	True if *lv* is greater than or equal to *rv*. For example, `nawk ' $1 >= "M" ' fruits` prints each line where the first field begins with a character that has a higher ASCII code than capital M. Such as N, O, P, Q, etc.
lv < *rv*	True if *lv* is less than *rv*.
lv <= *rv*	True if *lv* is less than or equal to *rv*.
lv == *rv*	True if *lv* is equal to *rv*. For instance, `nawk '$1 == "Oranges" ' fruits` prints each line where the first field equals the string "Oranges."
lv != *rv*	True if *lv* is not equal to *rv*.

Compound Patterns A compound pattern is an expression consisting of multiple expression patterns combined with Boolean operators. The Boolean operators are || (OR), && (AND), ! (NOT), and use of parentheses for grouping the operators. If the result of a compound pattern is true, the current input line is matched and the associated *action* is executed.

Boolean Operators The Boolean operators can be used to combine regular and relational expressions. The following table describes the Boolean operators.

Operator	*Function*
!	Negation. True if expression is NOT true. For example, `nawk '! /Apples/' fruits` prints each line that does not contain the string "Apples."
||	Logical OR. True if either expression is true. For instance, `nawk '$3 > "3,000,000" || $1 == "FRUIT"' fruits` prints the header line and each line with bushel counts of more than "3,000,000." Note 3,000,000 is a string, not a number, because of the commas.
&&	Logical AND. True if both expressions are true. For example, `nawk '$3 > "3,000,000" && NF == 5' fruits` prints the header line and each line containing exactly three fields.
( )	Perform comparisons enclosed together(for grouping). For instance, `nawk '( $1 == "Apples" || $1 == "Oranges" ) && ( NR == 4 )' fruits` prints lines where the first field is either "Apples" or "Oranges" and there are exactly four fields.

Range Patterns Range patterns contain two patterns separated by a comma (,). A range pattern matches each line from the occurrence of the first pattern to the occurrence of the second pattern. Both patterns may appear on the same line, thus only the one line is matched. If no input line

matches the first pattern, no lines are selected. If no input line matches the second pattern, all lines from the first pattern to the end of the file are selected. For example, the following command selects all of the lines from one that contains *Apples* to one that contains *Cherries*.

```
nawk '/Apples/,/Cherries/' fruits
```

ACTIONS The action part of an nawk program performs specified actions when a pattern is matched. If no action is specified the default *print* action is used. The default is the same as specifying { print }, which is equivalent to { print $0 }.

Comments The **nawk** command ignores any characters on a line following a # sign. Thus you can document your **nawk** programs using the number sign.

Constants Two types of constants are used by **nawk**, numeric and string. String constants are created by enclosing a sequence of characters in double or single quote marks. For example,

```
$1 == "FRUIT"
```

The word "FRUIT" is a string constant that is compared lexigraphically to field 1.

A numeric constant is an unquoted number. The number may be an integer, decimal, or floating point. All numbers are stored as floating points with the precision being machine dependent. Examples of number constants are:

```
3753
1.00E6
1.e8
2.78
```

Variables Variables are used to store dynamic values for later use. They can be used in patterns or actions. The type of a variable is not defined initially. Each time a variable is used, **nawk** decides what type it needs to use. It can be a number or a character type. There are three types of variables used in **nawk**: built-in, field variables, and user defined.

Built-in Variables These variables are set and updated by **nawk**. You can reset the values if you desire, but there is usually no need. All built-in variables have all uppercase names. The following table describes each of the variables.

Variable	*Description*
ARGC	The number of command line arguments
ARGV	The array containing the command line arguments
ENVIRON	Array of environment variables passed in from the shell. Refer to the env or printenv modules.
FILENAME	Current filename; contains the name of the current input file being read.
FNR	The number of the current record in the current file
NR	The number of current records read so far
NF	The number of fields on the current record
FS	Field separator; defines the characters that separate the current input line into fields. The default is spaces and tabs.
OFMT	The format used by **printf** for numbers. The default is %.6g.
OFS	Output field separator; defines the characters that separate the current output line into fields.
ORS	Output record separator; defines the characters that separate the output into separate records. The default is new-line.
RLENGTH	Length of the string matched by the **match** function

RS	Record separator; defines the characters that separate the input into separate records. The default is new-line.
RSTART	The starting position of the string matched by the **match** function
SUBSEP	The subscript separator. The default is \034 (^\).

Field Variables The fields of the current input record are assigned to the variables $*n* where *n* is the field number; field one is assigned to $1. The dollar sign informs **nawk** to interpret the variable as a field position. The field number may be a constant or a variable. You may reassign values to field positions and the record variable. The **record variable** is $0; it refers to the entire current input record.

Field Variable	*Description*
$0	Contains the entire current record (input line).
$*n*	Contains value of field *n* (column *n*). Where *n* is a numeric constant. For example, `print $1 " " $2` prints the first and second field separated by a space.

$*i*	Contains value of field *i*. Where *i* is a variable or arithmetic expression. For example,
	`for ( i = 1; i <= NF ; i++ )`
	`  print $i`
	prints each field on a separate line. You can also reference fields with arithmetic expressions; for instance,
	`print $(NF - 1)`
	prints the next to last field of the record.

User definable variables User definable variables are variables you assign values to as you desire. The variable names may contain letters, digits, and underscores. They must begin with a letter or underscore. You do not have to declare variables in **nawk**. A variable can be of string type or numeric type. If you reference a variable that has not been assigned a value, **nawk** uses the null string " " for a string value and zero (0) for a numeric value. You can use a string variable in a numeric expression and you can use a numeric variable in a string expression. **Nawk** converts the types as required automatically.

VAR	A variable named *VAR*
i=10	Assign the value of 10 to the variable i.
FRUIT="Apple"	Assign the string value of "Apple" to FRUIT.

Command Line Variable Assignment **Nawk** allows you to assign values to variables on the command line, thus permitting you to pass shell variables and values to the **nawk** script. These variables appear on the command line in the position of filenames. The syntax must be *var=value*. These assignment statements can be placed between filenames. This causes **nawk** to set the variable internally after one file has been read and before the next file is read. For example,

```
nawk -f progfile range=5,20 infile1 range=55,75 infile2
```

sets the variable *range* to 5,20 during the processing of file *infile1*, then resets the value of *range* to 55,75 for file *infile2*.

NOTE

Not all versions of nawk support the variable and filename interweaving.

Command Line Arguments The command line arguments are available inside the **nawk** script. The arguments are placed in a built-in array named ARGV. The number of arguments passed to the script are stored in the built-in variable ARGC. The ARGC variable is set to one more than the actual number of arguments, because the first arguments is always **nawk** and thus you must add one to the argument count. For example, in the following command ARGC equals 5.

```
nawk -f progfile aflag=1 bflag=0 cflag=1 infile
```

ARGV[0] contains **nawk**, ARGV[1] contains *aflag=1*, ARGV[2] contains *bflag=0*, ARGV[3] contains *cflag=1*, and ARGV[4] contains *infile*. This numbering and referencing of arguments scheme is compatible with the C programming language.

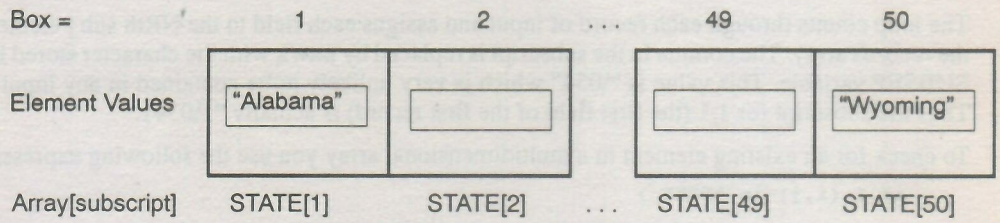
The *-fprogfile*, *'program,'*, *and -Fsep* are not considered arguments and, therefore, are not assigned to an element of the ARGV array.

Arrays An array is a data structure used to store related strings and/or numbers. Arrays are made up of elements. Each element is equivalent to a single variable. The difference is you address the array with one name and each element with a subscript. **Nawk** supports one-dimensional and multidimensional arrays.

One-dimensional Arrays **Nawk** provides one-dimensional arrays for storing data. This allows you to store multiple values in one variable name using a single subscript. For example, the following code assigns the *string* "Alabama" to the first element of the array named STATE and assigns "Wyoming" to the fiftieth element.

```
STATE[1]="Alabama"
STATE[50]="Wyoming"
```

To clarify an array structure, let's consider a very small post office. To keep it simple, let's say you only have one row of mail boxes and there are fifty mail boxes beside each other. The boxes are numbered from 1 to 50, and each box contains a piece of paper in it with the name of a state on it. For example,



The box numbers can be thought of as the subscript number used to address the elements of the array. The boxes themselves are like elements or cells of the array. Thus to refer to an element and retrieve the name of a state you simply type **STATE[*subscript*]**.

Arrays are declared automatically by **nawk** the first time you assign a value to an element of one. If no assignment has been made to an array element, the value returned is a null string " " or a numeric 0.

Subscripts The unusual aspect of arrays in **nawk** is the fact the subscripts are *strings*. They are not truly numbers. This allows you to reference elements in an array with string values. For example,

```
STATE["Alabama"]=1,900,200
```

assigns the value of 1,900,200 to the "Alabama" element of the STATE array. This is not the same as the STATE[1] element. To reference this element you simply use the "Alabama" string value.

Subscripts may be variables or constants. When you assign values to array elements, you must make sure you are consistent in using the same value to reference the correct element. For example,

```
STATE["Alabama"]=10
```

and

```
STATE[Alabama]=10
```

are not the same unless you have assigned the value of "Alabama" to the variable Alabama. Because the second statement is using a variable named Alabama which is set to null (" ") not the constant "Alabama."

To determine whether a subscript exists in an array you can use the following expression.

```
if ( "Wyoming" in STATE )
```

This statement checks the array STATE for a subscript named "Wyoming." It does not check for the value of "Wyoming" being in the array STATE.

CAUTION

When using numbers as subscripts you must remember ARRAY[1] references the same element as ARRAY["1"]. This is because **nawk** converts all subscript names to string values.

Multidimensional Arrays **Nawk** also supports multidimensional arrays. Actually it provides a simulation of multidimensional arrays. Since subscripts are strings, **nawk** concatenates the subscripts together to form one unique subscript to address an element in a one-dimensional array. The following example illustrates how you might utilize a multidimensional array.

```
for ( i = 1; i <= NF; i++ )
    records[NR,j] = $i
```

The loop counts through each record of input and assigns each field to the NRth sub j element of the *records* array. The comma in the subscript is replaced by **nawk** with the character stored in the SUBSEP variable. This value is "\034" which is very unlikely to be contained in any input text. Thus the subscript for 1,1 (the first field of the first record) is actually "1\0341."

To check for an existing element in a multidimensional array you use the following expression.

```
if ( (i,j) in ARRAY )
```

To loop through a multidimensional array you use the expression,

```
for ( i in ARRAY )
  split( i, ARRAY, SUBSEP )
```

Looping Through Arrays The previous expression can be used inside the *for* loop structure to loop through every subscript of an array. For example,

```
for ( state in STATE )
  print state "    " STATE[state]
```

prints the name of the state, three spaces, and the name value stored in the element with the subscript of *state*.

Deleting Array Elements It may be necessary at times to delete an array element from an array. For example,

```
delete STATE["Alabama"]
```

deletes the "Alabama" element from the STATE array.

Splitting Strings into Arrays The split function splits a string into fields and stores the separate fields in elements of an array. For example,

```
DATE="03/16/80"
split( DATE, BIRTH, "/" )
```

splits the value of the variable DATE (03/16/80) into three fields using the / as the field separator. It stores "03" in BIRTH[1], "16" in BIRTH[2], and "80" in BIRTH[3].

Arithmetic Operators The standard set of C arithmetic operators is supported by **nawk**. The following table defines each operator.

Operator	*Function*
+	addition
–	subtraction
*	multiplication
/	division
%	modulus
++	increment
––	decrement

Arithmetic Assignment Operators These operators perform faster arithmetic assignments than the conventional assignment operations. For example, to add the value of 10 to the variable X you normally type,

```
X = X + 10
```

Using an assignment operator you type,

```
X += 10
```

These two statements are equivalent. They both add 10 to the value of the variable X. The following table lists all assignment operators.

Operator	*Function*
+=	addition assignment; N = N + X or N += X
–=	subtraction assignment
*=	multiplication assignment
/=	division assignment
%=	modulus assignment

Logical Operators You can use logical operators to combine expressions to form logical expressions. There are three logical operators. They are listed in the following table.

Operator	*Function*
&&	Logical AND. For instance, `if ( ( NR > 10 ) && ( NF > 5 ) ) print` prints each line after line 10 that has five or more fields.
||	Logical OR. For example, `if ( ( NR < 10 ) || ( NF > 7 ) ) print` prints each line before line 10 or any line with more than seven fields.
!	Logical NOT. For instance, `if ! ( NF == 6 ) print` prints each line that does not have six fields.

The logical operators may be followed by a new-line. This allows for more readable statements when multiple logical operators are used in the same statement.

Increment and Decrement Operators As in the C programming language, **nawk** also provides the unary increment and decrement operators. You use these two operators to increment number variables and decrement them.

Operator	*Function*
++	Increment the value by 1. For example, `i++` is the same as i = i + 1.
--	Decrement the value by 1.

Conditional Expressions The conditional expression is used for simple if, then, else statements or conditions. The general format of a conditional expression is:

```
expr1 ? expr2 : expr3
```

where expr1 is evaluated and if true value of expr2 is used, else the value of expr3 is used. For example,

```
print $1 == 1 ? $2 : $3
```

prints the value of field two ($2) if field one ($1) is equal to 1, else the value of field 3 ($3) is printed.

Functions There are three different categories for functions. The built-in arithmetic functions, built-in string functions, and the user-definable functions. Functions are often referred to as subroutines in some programming languages. They allow you to write one set of statements and then reference them multiple times from different locations in the program. They also provide a means of making your program more understandable and readable.

Arithmetic Functions The arithmetic functions perform basic mathematical processing.

Function	*Value Returned*
atan2(*y*,*x*)	Returns the arctangent of *y*/*x* in the range –pi to pi.
cos(*x*)	Returns the cosine of *x*, with *x* being in radians.
exp(*x*)	Returns the value of e^x.
int(*x*)	Returns the integer value of *x*. Truncates *x*.
log(*x*)	Returns the natural (base *e*) logarithm of *x*.
rand(!)	Returns a random number *r*, 0 <= *r* < 1.
sin(*x*)	Returns the sine of *x*, with *x* being in radians.
sqrt(*x*)	Returns the square root of *x*.
srand(*x*)	Uses *x* as the new seed for the rand() function.

String Functions The string functions provide basic string manipulation functions. The following table describes the functions.

Function	*Value Returned*
delete array [*element*]	Deletes the specified element of the array.
gsub(*r*,*s*)	Substitutes *s* in place of *r* globally in $0 and returns the number of substitutions made.
gsub(*r*,*s*,*t*)	Substitutes *s* in place of *r* globally in sting *t* and returns the number of substitutions made.
index(*s*,*t*)	Returns the position where string *t* first occurs in string *s*. Returns 0 if *t* is not present.
length(*s*)	Returns the number of characters in *s*.
match(*s*,*r*)	Searches *s* for a substring matching *r*. The index of *r* is returned or 0. Sets the RSTART and RLENGTH variables.
split(*s*,*a*)	Splits string *s* into elements of array *a* based on fields defined by the field separator specified by FS.
split(*s*,*a*,*fs*)	Splits string *s* into elements of array *a* based on fields defined by the field separator *fs*.
sprintf(*fmt*,*expr-list*)	Returns the *expr-list* formatted according to the format string *fmt*.
sub(*r*,*s*)	Searches $0 for the longest substring matching string *r* and replaces with string *s*.
sub(*r*,*s*,*t*)	Search string *t* for the longest substring matching string *r* and replace with string *s*.
substr(*s*,*p*)	Returns the suffix of *s* starting at position *p*.
substr(*s*,*p*,*n*)	Returns the substring of *s* of length *n* starting at position *p*.

User-Defined Functions In addition to the built-in functions, **nawk** provides you with the ability to define your own functions. The format of a function is

```
function name( parameter-list ) {
    statements
}
```

Note that there cannot be any spaces between the *name* of the function and the left parenthesis. Functions can be defined anywhere a *pattern-action* statement is found. Functions provide a way to perform the same code multiple times without rewriting the code multiple times.

The **return** statement can be used to exit a function but is not necessary. The statements within the function are executed and when the last statement completes execution, the function is exited and control is returned to the line after the function call was made.

Functions may call themselves, thus being recursive. When a variable is used as an argument to a function, only a copy of the variable's value is passed to the function, thus the function cannot alter the value of the variable. The elements of an array can, however, be changed. New arrays can be created within functions and returned to the main program. If a variable is not listed in the variable list, then it is a global variable and its value can be manipulated within the function.

Input Control You can have **nawk** read input in several ways. The most common is from a file or from the standard input. For example,

```
nawk 'NF == 4' fruits
```

reads input from the file *fruits*. To have the data read in from the standard input you can either pipe the data to **nawk**, re-direct it from a file, or enter it from your keyboard. For example,

```
dbdownload | nawk -f dbreport1
```

could be used to take the output of a program and send if through an **nawk** program to format the output for a particular report.

Input Separators The input read by **nawk** is parsed into separate fields. The fields are separated based on the value of the FS variable. The default value of FS is a single space (" "). This informs **nawk** to consider spaces and tabs as field separators. The leading spaces and tabs are discarded and the first character that is not a space or tab begins the first field. The FS variable may contain regular expressions, allowing for any form of a field separator. To set the FS to a single space you use the regular expression "[ ]".

Input Records The input can also be parsed into separate records. By default, records and lines are synonymous. Because the default record separator variable RS is set to "\n," the RS variable can be reset to the null string "" to cause the record separator to be a blank line.

The getline Function The **getline** function provides a method of reading data from various sources. It can read input from the current input (pipe or data), from a specific file, or from an internal pipe. Depending on how you use **getline**, various variables are updated. The following table describes each format and how each variable is affected.

NOTE

The *cmd* in the following commands may be a constant or a variable.

Command	*Description*

getline	Reads the next line of the current input. The input line is assigned to $0, NF is set to the number of fields, NR and FNR are incremented. For example,
	`if ( $1 == "C" ) {`
	`  FIRSTLINE=$0`
	`  getline`
	`  SECONDLINE=$0`
	`}`

allows you to read a second line by choice instead of looping through the next entire iteration of the **nawk** program.

getline *var* — Reads the next line of the current input. The input line is assigned to variable *var*, NR and FNR are incremented.

getline < *file* — Reads the next line of *file*. The input line is assigned to $0 and NF is set. For example,

```
while ( getline < $2 )
    print
```

prints the entire contents of a file whose name was retrieved from the second field of the current input.

getline *var* < *file* — Reads the next line of *file*. The input line is assigned to variable *var*.

cmd | **getline** — Reads the next line of output from command *cmd*. The input is assigned to $0 and NF is set. When using pipes, getline reads one line of the output produced by the command each time the getline command is executed. An example of using this format

```
while ( "who" | getline )
    print "USER="$1 "      TTY="$3
```

reads the output of the UNIX who command and prints out the user name and tty number.

cmd | **getline** *var* — Reads the next line of output from command *cmd*. The input is assigned to the variable *var*.

Output Control There are two output producing statements. They are the **print** and **printf** commands. The **print** command is for simple output. The **printf** is for precise formatted output. The output can be redirected to files or pipes. Filenames and pipes may be constants, variables, or expressions.

NOTE

The *cmd* and *file* in the following commands may be a constant enclosed in double quotes or a variable containing the actual command.

The following table describes the **print** command.

Command	*Description*
print	Prints the entire current line ($0) to the standard output.
print *expr, expr ...*	Prints the expressions *expr* separated by the output separator, OFS. The default for OFS is a single space. The output from each print command is terminated with the output record separator, ORS. The ORS default is a new-line.
print *expr ..* > *file*	Prints the expressions to the file named *file*. The file is overwritten.
print *expr ..* >> *file*	Prints the expressions to the file named *file*. The output is appended to the end of the file.

print *expr* .. | *cmd* Prints the expressions to the standard input of the command *cmd*. For example,

```
for ( E in A )
  print A[E] | "sort > sorted 2>&1"
close("sort > sorted 2>&1")
```

prints the contents of each element of the array A and pipes the output to the UNIX sort command. The sort command redirects standard error and standard output to the file *sorted*. A pipe must be closed using the *close* command.

The **printf** command can send output to the same destinations as listed in the table describing the **print** command.

The **printf** is a much more sophisticated output statement. It generates formatted output. The general format of the printf statement is

```
printf( format, expr1, expr2, ..., exprn )
```

The parentheses are not necessary. The *format* argument is always required. The format argument specifies how to format each of the expressions *expr*. Each specification begins with a % sign. For example,

```
printf ("%-10s %5d %%%2s\n", $1, $2, $3 )
```

prints the first field left-justified in a 10-column output field; the second field is right-justified in a five-column output field; and the third field is left-justified in a two-column output field preceded by a percent sign.

The following table defines the formats available for use with the printf command.

Character	*Prints the Expression as*
c	A single ASCII character.
d	A decimal integer.
e	In exponential notation. The format is [-]d.ddddddE[+-]dd.
f	In exponential notation. The format is [-]ddd.dddddd.
g	Uses the shorter of e or f conversion and suppresses nonsignificant zeros.
o	An unsigned octal number.
s	A string of characters.
x	An unsigned hexadecimal number.
%	Prints the % sign. No related argument is needed.

Constants may be placed in the format string to print as themselves. Escape sequences may also be placed in the format argument. Refer to the section Escape Sequences for a list of valid escape sequences supported by the printf function.

Variable printf Format The format part of the **printf** statement may be a constant or a variable. The ability to use a variable allows you to have variable formatting. For example,

```
echo | nawk '{
for ( i = 1 ; i <= 20; i++ ) {
```

```
        format="%"i"s\n"
        printf(format, i)
    }}'
```

prints the number from 1 to 20 with 0 to 19 spaces preceding the number.

Closing Files and Pipes Once you open a file or pipe for output writing, it remains open for further writing until you close it. Each file and pipe are referenced by unique expressions. For example, if you write to a file

```
print > "mynawkoutfile"
```

then it remains open until you close it. To close the file you must refer to it by the exact same name. For example,

```
close("mynawkoutfile")
```

The closing of files is necessary if you write to a file, then want to read it in later in the same **nawk** program.

Control-Flow Statements **Nawk** provides many of the control-flow statements found in the C programming language. It has the *if-else* statement for decision making tasks. The looping statements of *while*, *for*, and *do* are supported, as well as various other statements. The following table briefly defines the control-flow statements provided in **nawk**.

Statement	*Description*
{ statements }	Groups a set of single statements into a group. These braces are inside the *action* braces. For example, `/John/ { if ( $3 > "18" ) { print adults++ } }` The outside set of braces are action braces. The inside set are grouping braces. They force all the statements to be executed if the *if* statement is true. Normally only the first statement after the *if* is executed.
break	Immediately exit present loop structure. For example, `i = 1 while ( i <= 100 ) if ! ( i % 2 ) print else if ( Array[i] == "END" ) break` This loop counts from 1 to 100 and displays even numbers only. If the string "END" is encountered in the Array, then the loop is exited immediately.
continue	Immediately start the next iteration of the enclosing *for*, *while*, or *do* loop.
do *statement* **while** (*expression*)	Execute the *statement*. If the *expression* is true, repeat the loop; if not, exit the loop.
exit	Exit processing the *pattern* and *action* commands and process the END section.

exit *expression*	Exit the **nawk** program and return the value of the *expression* as the program status code.
for (*expr*1; *expr*2; *expr*3) *statement*	Set *expr*1, loop while *expr*2 is true, and perform *expr*3 before the next iteration of the loop is executed. For example, `for ( i = 1; i < 100; i++ ) printf "%10d\n", i` prints the numbers from 1 to 100 in a 10-column output field.
for (*variable* **in** *array*) *statement*	Execute *statement* for each element in the *array* with *variable* set to the next subscript. For example, `split("1 2 3 4 5", NUMS) for ( N in NUMS ) print NUMS[N]` prints the numbers from 1 through 5 from array NUMS.
if (*expression*) *statement*	If *expression* is true, then execute *statement*. For example, `if ( $1 == 100 ) exit` causes the **nawk** program to exit after reading the first 100 lines of input.
if (*expression*) *statement*1 **else** *statement*2	If *expression* is true, execute *statement*1, else execute *statement*2
next	Start the next iteration of the main input loop. The next input record is read in and processed immediately.
while (*expression*) *statement*	If *expression* is true, execute the *statement* and repeat the loop. For example, `i = 100 while ( i >= 1 ) print array[i--]` prints the elements of an array in reverse order.
;	The empty statement

PROGRAM EXAMPLES The following are only the program parts of some useful **nawk** programs. You must add the **nawk** command and filenames to be processed to each of the following programs.

Command	*Description*
`'{ print NR:$0 }'`	# print line numbers(NR) and line($0)
`'{ print NR:NF }'`	# print line numbers and number of fields
`'{ print $1 }'`	# print first field of each line
`'{ print $2 $1 }'`	# print field 1 and 2 of each line, reversed
`'{ print $2" "$1 }'`	# same as above, separated by " "
`' s = s + $1 END { print s }'`	# sum up field one, only print the sum

`'{ if ( length > 72 ) {`
`print substr($0,1,72)" *"`
`print substr($0,73,132) }`
`else print $0 }'`
	# flag lines longer than 72 chars with a *
	and print rest of line on next line
	
	else print the line as normal

`'{print substr($0,index($0,$5),132)}'`	# print the line ($0) starting with column 5 continuing to the end of line

DIAGNOSTICS AND BUGS As with any programming language, there are some bugs. The most common problems appear to be the syntax of the *patterns* versus the *action*. Most of these problems are caused by users forgetting to insert a left or right brace ({}) or a double or single quote. Fortunately, **nawk** provides fairly good diagnostics when it aborts because of these problems.

Another area where problems arise is the handling of strings versus numbers. If you assign a string to a variable and then compare it to a number, it will not match. To convert a string to a number, add a zero. For example,

```
A="153"        # assigns the string "153" to variable A
A=A + 0        # converts A to a number
```

The same problem arises with numbers. You can convert a number to a string by adding a null string. For example,

```
N=153          # assigns the number 153 to variable N
N=N + ""       # converts N to a string
```

RELATED COMMANDS Refer to the **ed**, **ex**, **grep**, **sed**, and **ksh** commands described in modules 39, 43, 60, 117, and 71.

RELATED FILES By default, **nawk** reads from the standard input and writes to the standard output. If files are specified on the command line, then the files are read as input. You can use the **getline** command to read from specific files, pipes, or the current input.

The **print** and **printf** commands allow you to write to the standard output, specific file, and pipes. It is possible to read and write from various sources and destinations.

The one difference between **nawk** and most programming languages is the processing of input and output. The input is read automatically. The first line is read and all the *pattern-action* statements are checked for execution. If you want to read input from a specific file or pipe, you

must use a loop to read the entire file or read one line of the specified file each time you process a line of the normal input.

APPLICATIONS

The **nawk** utility is a very powerful and flexible tool. It provides the functions of **grep, egrep, sed, echo,** and various other commands all wrapped in one package. It is best suited for reading in tables of data, searching for specific information and performing string manipulations and numeric calculations, and displaying formatted output.

TYPICAL OPERATION

In this activity you use the **nawk** command to center a line of text. Begin at the shell prompt.

1. Type the following code in a shell script named *center*. You can use the **cat** command or your favorite editor (use vi). We use the **cat** command for simplicity. If you use the **cat** command, remember to press **Ctrl-D** to exit **cat**.

```
cj> cat > center
echo $* | \
nawk '{ for ( i = 1; i <= ( 40 - ( length / 2 )); i++ )
        printf " "
       print $0
}'
^D
```

2. Type **chmod 755 center** and press **Return** to change the modes on the new *center* command to -rwxr-xr-x. This allows all users to read and execute the script, but only you can write to it.
3. Now test the new shell script and **nawk** program. Type **center Centering Text on a line!** and press **Return**. Notice the output line is centered on the next line.

```
cj> center Centering Text on a line!
                                Centering Text on a line!
```

4. Turn to Module 25 (SV), Module 133 (BSD) to continue the learning sequence.

Module 7
banner (SV)

Russian Bandana 370 X005 1850 ← Spartan 1820

DESCRIPTION

The external **banner** command displays large letters on your terminal. Since it writes to the standard output you can redirect the output to a file or another command via a pipe. The banner command expands its arguments to your screen as large characters. Only ten characters can be printed on each line. Only three lines are displayed on your screen at one time.

COMMAND FORMAT Following is the general format of the **banner** command.

```
banner strings
```

Arguments The arguments to **banner** are strings of ASCII characters. Each string can be up to 10 characters long. Multiple strings may be listed. Each string is displayed on a line by itself. To display more than one string on a line, enclose the strings in double quotes ("string1 string2") to form one argument word.

FURTHER DISCUSSION You may wonder why a command like banner is provided. It seems more fun than work oriented. Maybe so, but it does have some utility value. It can be used to send messages to other users or print banner pages on printouts. The following example shows it being used to send another user a message.

```
cj> banner "Wake Up" > tmp
cj> write nancy < tmp
cj> rm tmp
```

Nancy's screen would look something like the following display:

```
Message from mylogin, tty11...

XX   XX          XXX                        XX  XX
XX   XX           XX                        XX  XX
XX   XX   XXXXX   XX  XX   XXXXX            XX  XX  XX XXX
XX X XX       X   XX XX   XX    X           XX  XX   XX  XX
XXXXXXX  XXXXXX   XXX     XXXXXXX           XX  XX   XX  XX
XXX XXX  X   XX   XX XX   XX                XX  XX   XXXXX
XX   XX  XXXXX X  XXX  XX   XXXXX            XXXX    XX
                                                    XXXX
```

It's always nice to keep a good friend from getting fired because she was sleeping on the job!

DIAGNOSTICS AND BUGS Strings that are longer than 10 characters are truncated, displaying only the first 10 characters.

RELATED COMMANDS Refer to Module 38 on **echo**.

RELATED FILES The **banner** command writes to the standard output. It only reads input from its command line arguments.

APPLICATIONS

The **banner** command is usually used to print large signs or letters to a printer or display them on a terminal. If is often used to produce a banner page on printouts sent to printers by the **lp** command. It is sometimes used to display the startup screen of a menu driven shell.

TYPICAL OPERATION

In this activity you use the banner command to display a simple message and then the time of day. Begin at the shell prompt.

1. Type **banner This is a banner message!** and press **Return**. The message immediately begins to display on your screen. Because this message is expanded into five large lines, the first few words scroll off your screen.
2. Type **banner "Same Line"** and press **Return**. Notice both words are displayed on the same line. If you use quotes around your strings, **banner** attempts to display all information on the same line.
3. Type **banner 'date +%H:%M'** and press **Return**. Your screen should look something like the following display.

```
cj> banner `date +%H:%M`
 XXXXX     XXXX          XXXX    XXXXXXX
XX   XX   XX  XX        XX  XX   X    XX
XX  XXX   XX  XX   XX       XX       XX
XX X XX    XXXX    XX     XXX       XX
XXX  XX   XX  XX           XX       XX
XX   XX   XX  XX   XX    XX  XX     XX
 XXXXX     XXXX    XX     XXXX      XX
```

4. Turn to Module 38 to continue the learning sequence.

Module 8
basename

Modern Physics
↳ Page 474 → UDA combination detail

DESCRIPTION

The external **basename** command parses the given path and returns only the filename. Any prefix ending in / (slash) is deleted from the path. If a second argument is present, it is used as a suffix to be deleted from the filename. For instance, the command

```
cj> basename /u1/ts/mylogin/junk.c \.c
junk
```

returns junk. The entire path (/u1/tech/mylogin/) and the .c suffix are removed.

The longest possible prefix is removed while the shortest possible suffix is deleted.

COMMAND FORMAT Following is the format of the **basename** command.

```
basename path [ suffix ]
```

Arguments Only two arguments may be passed to the **basename** command.

path The path that is parsed with only the filename being returned.

suffix A suffix string that is removed from the end of the filename.

RELATED COMMANDS Refer to the **dirname** command described in Module 35.

RELATED FILES The **basename** command writes to the standard output.

APPLICATIONS

The **basename** command is normally used as a substitution command within a shell script. It can be used to reduce a full path parameter or variable down to a single filename. It can also remove suffixes so a single parameter variable can be used to perform a task that uses the same filename with different suffixes.

TYPICAL OPERATION

In this activity you use the **basename** command to copy a file to a different directory and name. Begin at the shell prompt.

1. Type **cd /tmp** and press **Return**. This puts you in the /tmp directory just to force the use of paths in the following command.

A.D. → 1947 + '10 years' = 1957.
<*\\ March 21, 2001 → July 07, 2011. //*/>
August 15th, 00:00:01 A.M.

2. Type **cp $HOME/file1 'basename $HOME/file1 1'.bak** and press **Return**.

   ```
   cj> cp $HOME/file1 `basename $HOME/file1 1`.bak
   ```

   The $HOME/file1 file is copied to /tmp/file.bak. The cp command looks like cp $HOME/file1 file.bak after the basename command executes.

3. Type **rm file.bak** and press **Return**.
4. Type **cd** and press **Return** to return to your HOME directory.
5. Turn to Module 35 (SV), Module 91 (BSD) to continue the learning sequence.

Module 9
bg (csh, ksh)

DESCRIPTION

The internal **bg** command resumes stopped jobs by placing them in the background. The **bg** command is part of the job control feature of the C shell and the Korn shell. Without arguments, **bg** refers to the current job. The current job is the last job you **stopped** or **suspended**.

COMMAND FORMAT Following is the general format of the **bg** command.

```
bg [ %job ... ]
```

Arguments The following arguments may be passed to the **bg** command.

%job A job number associated with a process. If the interactive option is on, *job* may be one of the following:

%num	A job number for an associated process.
%string	The job whose name begins with *string*.
%?string	The job whose name contains *string*.
%+	
%%	Current job; the last process you stopped or suspended.
%-	Previous job; the process before the current job.

RELATED COMMANDS Refer to the **fg**, **jobs**, **kill**, and **suspend** commands described in modules 51, 68, 70, and 129.

APPLICATIONS

The **bg** command is used to restart suspended jobs in the background. You can suspend a job running in the background or one running in the foreground. This is handy if you are running a command and you need to perform some other function. For example, let's assume you are running a long process in background. You realize you need to create a file the process requires. You can suspend the job, create the file, then restart the process by using the **bg** command.

TYPICAL OPERATION

In this activity you use the **bg** command to restart a process in the background. Begin at the shell prompt. Before you start you may wish to check your suspend character, type stty -a or stty everything (BSD) and see to what character the suspend (susp) character is set.

1. Type **cd /** and press **Return**.
2. Type **du -s > /dev/null** and press **Return**. This produces a disk usage list and writes the output in the UNIX trash can device.

3. Press **Ctrl-Z** to suspend the du job.

```
cj> du -s > /dev/null
^Z
[1] + Stopped                    du -s > /dev/null
```

4. Now type **jobs** and press **Return** to display all background and suspended jobs.

```
cj> jobs
[1] + Stopped                    du -s > /dev/null
```

5. Type **ps** and press **Return** to display the time used by the du process. If you want, wait 15 or 20 seconds and repeat this step. Notice the time used by **du** has not increased. This is because the job is suspended.

6. Type **bg %1** to start the **du** job running in background. Your screen should resemble the following,

```
cj> bg %1
[1]du -s > /dev/null
```

   The format of the job display from **bg** may vary from system to system.

7. Type **ps** and press **Return** to display the time used by the du process. Notice that the time used has increased since the last time you performed a **ps**. This is because the job is running in the background.

8. Terminate the background job by typing **kill %1** and pressing **Return**. The following message is displayed,

```
cj> kill %1
[1] + Terminated                 du -s > /dev/null
```

9. Turn to Module 51 to continue the learning sequence.

Module 10
cal

DESCRIPTION

The external **cal** command generates a simple calendar and writes it to the standard output. The default output for **cal** is the current month. Using the arguments listed, a specific year or year and month can be created. Valid years are from 1 to 9999. Valid months are in the range of 1 through 12. The calendar produced is the English/United States standard.

COMMAND FORMAT Following is the general format of the **cal** command.

```
cal [ [ month ] year ]
```

Arguments The following list describes the arguments that may be passed to the **cal** command.

year Produces calendar for the given year.

month Produces a calendar for the specified month of the given year.

Remember that a single argument will return the calendar for the given year. The year must be a complete number, i.e., if you want to create the calendar for 1980, you must enter 1980. If you just enter 80 the calendar for 80 A.D. is displayed.

FURTHER DISCUSSION A strange calendar is produced for September 1752. Because leap years were not accounted for until this time, 11 days were skipped to make the correct adjustment. To display this calendar, type cal 9 1752 and press Return. Your screen should resemble the following display:

```
cj> cal 9 1752
  September 1752
Su Mo Tu We Th Fr Sa
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

DIAGNOSTICS AND BUGS The assumption is made that each year always starts in January even though this is historically incorrect.

The command format "cal 45" will return the calendar for the ealy Christian era, not the calendar for 1945.

RELATED COMMANDS Refer to the **date** and **calendar** commands described in modules 30 and 11.

RELATED FILES The **cal** command writes to the standard output.

APPLICATIONS

The **cal** command is used to generate human readable calendars for everyday use. It is a useful on-line utility that saves you from having to look up a day or date on a paper calendar.

TYPICAL OPERATION

In this activity you use the **cal** command to create a calendar for an entire year and one for a single month. Begin at the shell prompt.

1. Type **cal 1752** and press **Return**. Notice a calendar for each month of 1752 is generated.

   The month of September is correct. These days were skipped to correct the fact that leap years were not being recognized every four years.

2. Type **cal 1 1989** and press **Return**. Your screen should resemble the following display:

```
cj> cal 1 1989
    January 1989
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

3. Turn to Module 11 to continue the learning sequence.

Module 11
calendar

DESCRIPTION

The external **calendar** command provides a daily reminder service. It searches for strings anywhere in the "calendar" file that match today's or tomorrow's date.

COMMAND FORMAT Following is the general format of the **calendar** command.

```
calendar [ - ]
```

Arguments The following argument may be passed to the **calendar** command.

- If present, calendar performs its functions for every user who has a "calendar" file in his login(usually $HOME) directory. The output from **calendar** is sent to the user via **mail**.

FURTHER DISCUSSION The following list illustrates the most common forms of date that **calendar** searches for in the file:

Format	*Example*
Mon. DD	Jan. 21
Month DD	January 21
MM/DD	01/21

The **calendar** command does NOT recognize the following formats:

DD Month	21 January
DD/MM	21/1

The concept of tomorrow extends across weekends. For example, if you run **calendar** on a Friday, it will display any lines containing Monday's date.

If the environment variable DATEMSK is set, calendar will use the format strings in the specified file based on the values of the environment varialbes LANG and LC_TIME. Refer to the module on **date** for a description of available format strings.

DIAGNOSTICS AND BUGS Your calendar file must be readable public information for your reminder service to work properly. That is to say the permissions on your calendar file must at least be r--r--r--.

The extended idea of tomorrow does not account for holidays.

RELATED COMMANDS Refer to the **cal**, **date** and **mail** commands described in modules 10, 30, and 86 respectively.

RELATED FILES The **calendar** command writes to the standard output. The following list describes the files that calendar uses to perform its tasks.

/usr/lib/calprog	Used to figure out the correct dates for today and tomorrow.
/etc/passwd	System passwd file used to find each users login (HOME) directory.

APPLICATIONS

The **calendar** program is used to provide a daily reminder service. If you need to remember certain appointments or assignments on a daily basis, the **calendar** command is the answer. It can be run by the system for you or you can execute it for yourself as needed.

TYPICAL OPERATION

In this activity you use the **calendar** command to display lines from the calendar file. The first step is to create a calendar file. Begin at the shell prompt.

1. Type **cat > calendar** and press **Return**. You may use an editor if you wish.
2. Type the following lines of text and press **Ctrl-D**. This creates a calendar file.

   NOTE

   Replace the following Jan. 17 and 21 dates with tomorrow's date and some other date.

```
cj> cat > calendar
Jan. 17 - Take the car at lunch for an oil change.
Jan. 21 - Begin restructuring of all UNIX workstations.
^D
```

3. Now that you have a calendar file to use, type **calendar** and press **Return**. Your screen should resemble the following display.

```
cj> calendar
Jan. 17 - Take the car at lunch for an oil change.
```

4. Turn to Module 30 to continue the learning sequence.

Module 12
cancel (SV)

DESCRIPTION

The external **cancel** command removes or cancels print requests made with the **lp** command. The **cancel** command allows you to:

- Remove requests from a printer request queue.
- Stop a request currently printing on a printer.

The **cancel** command is used in conjunction with the **lp** and **lpstat** command. You may need to use the **lpstat** command to decide which request you want to cancel. When you cancel a request that is printing, the printer is signaled to stop printing. The request is removed and the next request is sent to the printer.

COMMAND FORMAT Following is the general format of the **cancel** command.

```
cancel [ request-ID ] [ printer ]
cancel [ -u login-ID ] [ printer ]
```

Options The following option may be used to control how **cancel** functions.

-u *login-ID* Cancel all requests for the specified user. Where login-ID can be the username or login-ID.

Arguments The following list describes the arguments that may be passed to the **cancel** command.

request-ID The request is removed from the request queue or stopped if currently printing.

printer Stops the current request printing on the specified printer. The request must belong to you.

RELATED COMMANDS Refer to the **lp** and **lpstat** commands described in modules 79 and 83.

RELATED FILES The requests that **cancel** removes are actually files stored in the /usr/spool/lp/request directory.

APPLICATIONS

You can use the **cancel** command to stop the printing of a request or remove the request from the print queue. It is useful when you have sent a print request to the wrong printer or a printer which is no longer in service.

TYPICAL OPERATION

In this activity you use the **cancel** command to remove a request submitted by the **lp** command. Begin at the shell prompt.

1. Type **ls -x /bin | lp** and press **Return**. Notice the request ID that **lp** returns to your screen.

```
cj> ls -x /bin | lp
request id is dp-1492 (1 file)
```

2. Now type **cancel dp-1492** to cancel the request. Replace the dp-1492 with your request ID.
3. Turn to Module 83 to continue the learning sequence.

Module 13
cat

DESCRIPTION

The external **cat** command reads each file in the argument list and displays the data to the standard output. It can read data from files or from the standard input. It only writes to the standard output. By using the shell redirection capabilities you can use **cat** to combine multiple files into one large file, thus the name con*cat*enate.

For such a simple command it is probably one of the most useful. Some of the uses of **cat** include:

- Display contents of ASCII files.
- Display control characters in a readable format.
- Make a copy of a file.
- Combine multiple files into one file.
- Copy input from your keyboard to a file.
- Send output to a pipe; useful for multiple file input.
- Combine input from your keyboard and a file into one output stream.

WARNING

The command **cat f1 f2 > f1** destroys the contents of file f1. The reason is that the shell performs redirection to f1 before **cat** opens and reads f1. So the contents of f1 are gone before **cat** has a chance to read the file.

COMMAND FORMAT Following is the general format of the **cat** command.

```
cat [ - ] [ -su ] [ -v [ -et ] ] [ file_list ]
```

BSD (Berkeley)

```
cat [ -u ] [ -n ] [ -s ] [ -v ] [ file_list ]
```

Options The following list describes the options used to control how **cat** functions.

–	Read from the standard input, treated as one input file. Terminate cat by entering a Ctrl-D from your terminal keyboard.
–s	Error messages for nonexistent files are suppressed.
–u	Unbuffered raw output, the data is written to the standard output character-by-character. Normally the output is buffered; written in blocks of characters.

–v Prints nonprinting characters in a visible form. Control characters are printed as ^X (control-X). Non-ASCII characters (high bit set) are printed as M-x; x is the character represented by the 7 low order bits.

–e Prints a $ (dollar sign) to represent the new-line character, i.e., the end of the line.

–t Prints a ^I (Ctrl-I) in place of tabs.

BSD (Berkeley)

–n Displays line numbers before each line of text. This is useful for program listings and other types of data where you need to know the line numbers.

–b Omits line numbers on blank lines. Implies the -n option. The blank lines are displayed without preceding line numbers.

–s Removes multiple adjacent empty lines. The output is displayed single spaced. If a line contains spaces and/or tabs it is not considered to be empty.

Arguments The following list describes the arguments that may be passed to the **cat** command.

file_list One or more files, **cat** will attempt to read in sequential order and write to the standard output.

If no arguments are specified, **cat** reads from the standard input.

FURTHER DISCUSSION The output of **cat** is displayed on your terminal screen. If the output is longer (has more lines) than your terminal screen, it scrolls upward until the last line is displayed. Your terminal prompt is then redisplayed. If the output scrolls up too rapidly and you need to stop the output from scrolling, press Ctrl-S. When you want to continue with more output, press Ctrl-Q. If you set **stty ixany**, you do not have to use Ctrl-Q; any character will restart your output.

Displaying a File To display a file you simply type a command like:

```
cat myfile
```

The contents of *myfile* are displayed on your terminal screen.

Displaying Multiple Files To display more than one file you simply name more than one file on the command line. For example,

```
cat myphonelist yourphonelist companyphone
```

displays the contents of all three files. There is no notification of when a file ends and the next one begins. The input is combined into one contiguous output stream.

Copying a File You can make a copy of a file using **cat**. Simply redirect the output to a file. For example,

```
cat original_file > new_file
```

copies the *original_file* into the file named *new_file*. The command is a combination of **cat** reading the source file and the shell redirecting the output to a file.

Combining Multiple Files You can combine several files into one file using the **cat** command and shell redirection. For example,

```
cat myphonelist yourphonelist companyphone > phonelist
```

combines the contents of three files into one new file. For more information on redirection refer to Module 67.

Creating a File from Keyboard Input You can also use **cat** to create a new file by typing the input from your keyboard. The following is a brief example:

```
cj> cat > newfile
This is a new file created by cat and shell redirection.
This is a simple way to create new files of text.
But remember you can only edit the current line.
You can erase previous characters or the entire current line.
To erase the previous character use the character defined as erase
by stty. To kill the entire current line use the kill character defined
by stty. To stop input and return to the shell you must press a Ctrl-D
at the beginning of a blank line. A Control-D is the end-of-file mark.
Press a Control-D on the next line.
^D
```

You have now created a file named "newfile." If you want, you can display it to your terminal by typing **cat newfile**. You can also edit it using an editor if you like.

DIAGNOSTICS & BUGS The command **cat file1 file2 > file1** will erase the contents of file1. Be careful using shell redirection with **cat**.

RELATED COMMANDS Refer to modules 24, 103, and 105 for descriptions of the **cp**, **pg**, and **pr** commands.

RELATED FILES The **cat** command is a filter; therefore, it can read the standard input and write to the standard output.

APPLICATION

The **cat** command can be used to concatenate files together, display the contents of files, and copy files. The concatenation of files is joining one file to the end of another. Displaying files is sending the data to your terminal screen. Copying files is done in conjunction with the shell's redirection capabilities.

Other commands have been designed to provide enhanced functions of **cat**. To display files use the **pg** (SV) or **more** (BSD) command. The **cp** command is designed specifically for copying files.

TYPICAL OPERATION

In this activity the **cat** command is used to display and create files using **cat**. Begin at the shell prompt.

1. Type **cat file1** and press **Return**. Notice that the contents of *file1* are displayed to your screen.
2. Type **cat > file1** and press **Return**. The existing *file1* is overwritten.

3. Type the following text; press **Return** after each line. Press **Ctrl-I** and **Ctrl-L** in the text below; do not type "Ctrl-I" and "Ctrl-L." Press **Ctrl-D** to stop input to the **cat** command.

```
cj> cat > file1
This text is being read by cat from my terminal keyboard.
Cat will read the text as it is typed and create file1.

With a blank third line!
^I ^L Just a few control keys for demonstration.
^D
```

4. Type **cat -vte file1** (use **-v -t -e** on BSD systems) and press **Return**. Notice the **$** (dollar signs) and the control characters (^I, ^L). The -vte option informs **cat** to display control characters in a readable format.

```
cj> cat -vte file1
This text is being read by cat from my terminal keyboard.$
Cat will read the text as it is typed and create file1.$
$
With a blank third line!$
^I ^L Just a few control keys for demonstration.$
```

5. Type **cat file1 - file1 > file2** and press **Return** to make two copies of file1 in file2 with input from your keyboard placed in between the two copies.
   a. Now type ===== and press **Return**.
   b. Press **Ctrl-D** to end the keyboard input.
6. Now display file2 by typing **cat file2** and pressing **Return**. Your output should resemble the following display. Notice the five equal signs (=====) between the two files.

```
cj> cat file2
This text is being read by cat from my terminal keyboard.
Cat will read the text as it is typed and create file1.

With a blank third line!
        Just a few control keys for demonstration.
=====
This text is being read by cat from my terminal keyboard.
Cat will read the text as it is typed and create file1.

With a blank third line!
        Just a few control keys for demonstration.
```

7. Type **ls** and press **Return** to display the files in your current directory. Your display should resemble the following:

```
cj> ls
bin     calendar  db   file1     file2     letters   myfile
```

8. Turn to Module 63 to continue the learning sequence.

Module 14
cd

DESCRIPTION

The internal **cd** command changes your present working directory. There are two forms of the change directory command. The first changes you to a new destination based on the single argument given. The second performs a string substitute on the current path and then changes directories. The **cd** command lets you do the following:

- Change working directories
- Change to last previous working directory
- Use a string substitution in the current path to change to a similar directory path
- Use tilde substitution to change to $HOME/subdirectories
- Use tilde substitution to change to another user's $HOME/subdirectories

The tilde substitution, - option, path substitution, and CDPATH does not apply to the **sh**.

COMMAND FORMAT Following is the general format of the **cd** command.

```
cd ~
cd [ directory ]
cd old new

chdir ~
chdir [ directory ]
chdir old new
```

Options The following option may be used to control how **cd** functions.

- The present working directory is changed to the previous working directory. The shell maintains an OLDPWD variable to know the previous directory path.

Arguments The following list describes the arguments that may be passed to the **cd** command.

directory Specifies the new present working directory. The default is $HOME if you do not specify a directory. See tilde substitution in the following section.

old The old string searched for in the current working directory's path that is changed to the new string.

new The new string that replaces the old string in the current working directory's path.

FURTHER DISCUSSION Using the first form of **cd**, you may use an absolute path or a relative path. An absolute path begins with a / (slash), a relative path begins with a directory name or a . (dot). Either form can be used to achieve the same result. Your decision should be the shortest path to reach your destination or the path syntax you understand best. The following examples help

clarify the use of absolute and relative paths. Both commands will place you in the same directory from your HOME directory. Assume your current directory is /u1/ts/mylogin.

```
cj> cd /u1/ts/nancy          # absolute path to user nancy
cj> cd                       # return to $HOME
cj> cd ../nancy              # relative path to user nancy
```

If no arguments are given, **cd** assumes the HOME variable is the desired directory.

```
cj> cd                       # same as cd $HOME
```

If the - (hyphen) option is present, *cd* changes to the previous directory.

```
cj> cd -                     # same as cd $OLDPWD, back to ../nancy
```

Tilde Substitution If the ~ (tilde) option is present, **cd** places your HOME path in front of any path you provide after the ~. If the ~ is followed by a string, then the shell searches the /etc/passwd file for a matching user name. If one is found, that user's HOME path is used. For example:

```
cj> cd ~/db                  # cd to $HOME/db
cj> cd ~nancy/bin            # cd /u1/dvlp/nancy/bin
```

The first example uses your HOME directory($HOME/db), whereas the second uses nancy's HOME directory (field 5 of /etc/passwd).

CDPATH If a single directory name is specified, **cd** searches the directories listed in the CDPATH variable. Each directory listed in CDPATH is searched for a subdirectory that matches the specified argument. The first match is used as the new directory.

The CDPATH variable is a colon-separated list of directory paths that **cd** searches for a matching subdirectory. The default CDPATH is null, specifying the present working directory. Therefore, if you do not have CDPATH set, **cd** will search only the current directory for the directory name given as an argument. If you set CDPATH=:/usr:/usr/spool, **cd** will look for the requested directory in your HOME directory first, then check the /usr directory, and then search /usr/spool. For example:

CAUTION

The current directory must be the first path defined in the CDPATH variable. The current directory is defined as CDPATH=: or CDPATH=.:. If you do not specify the current directory first, **cd** will complain about not being able to find a directory in the current directory if the directory is not specified in the CDPATH variable.

```
cj> CDPATH=:/usr:/usr/spool
cj> export CDPATH
cj> cd bin
cj> pwd
/usr/bin
```

would **cd** you to the /usr/bin directory. To **cd** to your db directory, you would just type

```
cj> cd db
cj> pwd
/u1/ts/mylogin/db
```

Your new present working directory would be /u1/ts/mylogin/db.

The second form of the **cd** command is to have a string of the current path substituted with a new string. The following scenario reveals how **cd** handles substitutions within paths. You may want to create these directories and then try these commands. Type mkdir C/src sh/src C/src/shtools sh/src/shtools and press Return.

```
cj> cd C/src/shtools
cj> pwd
/u1/ts/mylogin/C/src/shtools
cj> cd C sh                    # substitute sh in place of C
cj> pwd
/u1/ts/mylogin/sh/src/shtools
```

The substitution command is useful to switch between directories sharing almost the exact same long path names.

NOTE

The **cd** command cannot be executed from a restricted shell.

DIAGNOSTICS AND BUGS The following message may be returned if **cd** cannot perform the requested directory change.

"*directory*" bad directory	The directory is not readable and executable by the requesting user or it does not exist.

RELATED COMMANDS Refer to the **pwd** command described in Module 111.

RELATED FILES The **cd** command does not display a response unless a problem is encountered. If a problem is reported, it is written to the standard error.

APPLICATIONS

The **cd** command changes you from your present working directory to a new directory. It is used to move around a file system to perform tasks without having to specify entire paths.

If you are in a directory with the files you need, you can refer to the files as needed. If you are not in the same directory, then you have to specify a path each time you want to access the files. Therefore, **cd** is a handy tool to move you where you need to be in the file system.

TYPICAL OPERATION

In this activity you use the **cd** command to move around to various directories on the system. Begin at the shell prompt.

1. Type **cd /usr/spool/uucppublic** and press **Return**. You are now in the uucp spool directory.
2. Type **cd ~/bin** and press **Return** to change directories to your $HOME/bin directory.
3. Type **cd ../db** and press **Return** to change to your $HOME/db directory.
4. Type **cd** and press **Return** to change directories back to your HOME directory.
5. Turn to Module 111 to continue the learning sequence.

Module 15
chfn (BSD)

DESCRIPTION

The external **chfn** command allows you to change the information field in the system password file /etc/passwd. An alternative command to **chfn** is **passwd -f**. Either of these commands displays the current information stored in the fifth field of the /etc/passwd file. This information is referred to as the GECOS information. The fifth field consists of several subfields, which are comma separated. You are prompted for new information to replace each of these subfields. The fifth field usually includes the following subfields for each user:

+ "real life" name
+ office room number
+ office phone number
+ home phone number

The prompt contains the existing value of the subfields within square brackets ([subfield value]). The existing value is retained by simply pressing Return. If you want the field to be empty, type the word "none" at the prompt. Phone numbers may or may not contain hyphens.

It is advisable to run the finger command after running **chfn** or **passwd -f**.

The super-user can change any user's GECOS information. You can only change your own GECOS information as a normal user. It is a description of the user of the account. The **finger** program displays this information.

COMMAND FORMAT Following is the general format of the **chfn** command.

```
chfn [ user_name ]
```

or

```
passwd -f [ user_name ]
```

Options The following list describes the option used to control **passwd** to function like **chfn**.

-f Informs **passwd** to execute the **chfn** function to change the GECOS information stored in the /etc/passwd file.

Arguments The following argument may be used to change how **chfn** functions.

user_name Causes **chfn** to change the information for the specified user instead of the user executing **chfn**.

RELATED COMMANDS Refer to the **passwd** and **finger** commands described in modules 100 and 56.

RELATED FILES The **chfn** command updates the information in the fifth field of the /etc/passwd file. The fields in the /etc/passwd file are separated by colons.

APPLICATIONS

You will need to use the **chfn** command to keep the information pertaining to your office room number and phone numbers updated.

TYPICAL OPERATION

In this activity you use the **chfn** command to change your office phone number. Begin at the shell prompt. The information in the prompts will be different for each user's login. Use your actual office phone number in the following command.

1. Type **chfn** and press **Return**. You will be prompted as follows; press **Return** after each prompt that you do not want to update.

```
cj> chfn
Name [David Hancock]: <CR>
Room Number (Exs: 597E or 197C) [69]: <CR>
Office Phone (Ex: 1632) [474]: 483<CR>
Home Phone (Ex: 987532) [555-1212]: none<CR>
```

2. Turn to Module 38 to continue the learning sequence

Module 16
chgrp

DESCRIPTION

The external **chgrp** command changes the group ID of a file or directory. The group ID is used to inform the system which group has group access permissions to a file. If you are a member of the group that the group ID of a file is set to, then the group access permissions apply to you when you try to access a file.

Each file on the system is identified by a group ID. Members of that group have access to the file as specified by the group permissions. By using the **chgrp** command you can change which group the access permissions affect.

COMMAND FORMAT Following is the general format of the **chgrp** command.

```
chgrp [ -hR ] group file_list
chgrp [ -hR ] group directory_list
```

BSD (Berkeley)

```
chgrp [ -fR ] group file_list
```

Options The following options may be used to control how **chgrp** functions.

-h — If the file is a symbolic link, change the group of the symbolic link instead of the file pointed to by the link.

-R — Forces **chgrp** to recursively descend the directory trees listed, changing the group ID of files and directories. If a symbolic link is encountered, its group ID is changed but it is not traversed.

BSD (Berkeley)

-f — If *chgrp* fails to change the group or incurs errors, it does not report them to your screen.

Arguments The following arguments may be passed to the **chgrp** command.

group — The group you wish to permit access to your files or directories. You may provide a group name or a group ID for group. For example, your group name is "ts" and your group ID is "40." 13th OCTOBER 2015.

file_list — One or more files to change the group ID so a different group can access the files.

directory_list One or more directories to change the group ID so a different group can access the files.

FURTHER DISCUSSION Usually the system administrator separates users into logical groups based on functionality of departments within the company or project. The /etc/passwd file sets your group ID to a number. The /etc/group file contains this number and its corresponding group name. The file also contains a list of users who may be or become members of that group.

For instance, if you're a member of the ts group when you log in, you are listed in the /etc/passwd file with group ID 40 (fourth colon-separated field). In the /etc/group file there is a line containing 40 (third colon-separated field) and the name of the group (first field). Your login may be listed on this line (fourth field) but it is not required since this is your primary login group. If you are listed on another line, you can also become a member of that group by using the **newgrp** command.

BSD (Berkeley)

The groups in BSD function differently from the groups in System V. In System V you are in one group at a time. For example, if you are in the ts group when you log in, then permissions on files and directories with ts group IDs apply to your login. Even if you are listed in another group, only your current group ID is checked. Whereas in BSD if you are listed in several groups in the group file, then you are a member of those groups at all times.

If the set-user-ID or set-group-ID mode of a file is turned on, **chgrp** will clear the mode from the file access permissions. If you are super-user the modes are not cleared.

NOTE

You must be the owner or super-user of the file to be able to change its group ID.

RELATED COMMANDS Refer to the **chmod** command described in Module 17 and **chown** described in Module 18.

RELATED FILES The **chgrp** command looks up the related group ID for a given group name in the /etc/group file.

APPLICATIONS

The **chgrp** command is used to change the group ID of a file or directory. It allows you to change which group may access your files. There may be certain sets of files that you need to share with a documentation group. By using the **chgrp** command you can change the group ID so users in the documentation group can access the files and directories.

TYPICAL OPERATION

In this activity you use the **chgrp** command to change the group of a file using the group name and then back to its original group using the group ID number. Begin at the shell prompt.

1. Type **chgrp dvlp db** and press **Return**. The development group can now access your db directory. You may have to select a different group from the /etc/group file in place of dvlp. Remember, you have to be a member of the group to change the group ID to a different group.

2. To see the new group type **ls -l db** and press **Return**. Your screen will look like the following display.

```
cj> ls -l db
-rwxr-xr-x  1  mylogin   dvlp     1 Jan 24 07:15 db
```

3. Type **chgrp 40 db** and press **Return** to change the group for db back to ts. Most users use the name of the group, but you can use the group ID. Again 40 is assumed to be the group ID for your primary group.
4. Turn to Module 17 to continue the learning sequence.

Module 17
chmod

DESCRIPTION

The external **chmod** command allows you to change the modes of a file or directory. The modes are the access permissions controlling who can and cannot access the file or directory. File modes control file attributes such as:

- Who can read from a file
- Who can write to a file
- Who can execute a file
- If a file allows you to assume its user ID during execution
- If a file allows you to assume its group ID during execution
- If a file is to remain in memory after execution
- If a file is to be locked while being accessed by a user

The **chmod** allows you to change these attributes using:

- Absolute modes; all number arguments
- Symbolic modes; who strings, operators, and permission strings

The modes are classified into four categories. The first is the special modes. The last three are the permissions for different sets of users:

- The owner of the file
- The group associated with the file
- All other users on the system

COMMAND FORMAT Following is the general format of the **chmod** command.

```
chmod [ -R ] absolute_mode file_list
chmod [ -R ] absolute_mode directory_list

chmod [ -R ] symbolic_mode file_list
chmod [ -R ] symbolic_mode directory_list
```

BSD (Berkeley)

```
chmod [ -fR ] absolute_mode file_list
chmod [ -fR ] absolute_mode directory_list

chmod [ -fR ] symbolic_mode file_list
chmod [ -fR ] symbolic_mode directory_list
```

Options The following list describes the options that are used to control how **chmod** functions.

absolute_mode A three- or four-digit octal number describing the permissions. See the following section on Absolute Modes.

symbolic_mode A string made up of three parts describing the permissions. The first part is the "who string"; who is affected. The second is an "operator string"; add, remove, or absolutely assign permissions. The third is the "permission string"; what are the permissions for the file or directory. See the following section on Symbolic Modes.

-R Forces **chmod** to recursively descend the directory trees listed, changing the group ID of files and directories. If a symbolic link is encountered, its group ID is changed but it is not traversed.

BSD (Berkeley)

-f If **chmod** fails to change the group or incurs errors, it does not report them to your screen.

-R Same as SV.

Arguments The following list describes the arguments that may be passed to the **chmod** command.

file_list One or more files you want to give new modes.

directory_list One or more directories you want to give new modes.

FURTHER DISCUSSIONS The **chmod** command changes which users can read, write, and execute a file. The modes are separated into four classes:

special	Special permissions
user	The owner of the file or directory
group	The users in the group that the file is associated
other	All other users on the system

Each class of users has a set of modes controlling their access permissions.

The owner and super-user are the only people who can change the permissions of a file or directory. To set the set-group-ID the file must have the same group ID as your current group ID.

NOTE

The super-user can access any files or directories regardless of file permissions. This should be considered when storing personal or private corporate information. If you do not want the system administrator to access the data, you might want to consider using the **crypt** command.

Absolute Modes The absolute mode is a three- or four-digit octal number. Each position of the number equates to a special meaning or a class of users. Absolute modes are a shorthand for changing permissions. Therefore, they are terse and more difficult to remember and use.

The first digit is not required unless you wish to change the special modes. Thus you are allowed to use a three-digit number when you are only changing file permissions relating to reading, writing, and executing. The following list relates each position to what it does.

X000	Special meaning
0X00	Modes for the user class
00X0	Modes for the group class
000X	Modes for the other class

The following table defines each possible value that may be used in the absolute modes:

Special Modes

4000	Set the user ID at execution time.
20#0	Set the group ID or file locking at execution time. If # is 7, 5, 3, or 1, set the group ID. If # is 6, 4, 2, or 0, enable mandatory file locking.
1000	Sticky bit is turned on. Informs the operating system to keep the program text loaded in memory after execution. Useful for programs that are executed many times each day.

User Modes

0400	The owner has read permission.
0200	The owner has write permission.
0100	The owner has execute permission. If permission is for a directory, then owner has search permissions.

Group Mode

0040	The members of a group have read permissions.
0020	The members of a group have write permissions.
0010	The members of a group have execute and search permissions.

Others Mode

0004	All users other than the owner and group members have read permissions.
0002	All users other than the owner and group members have write permissions.
0001	All users other than the owner and group members have execute and search permissions.

To obtain the set of permissions desired, add up the numbers for a given class. For example, if you want read and write permissions, you would add 4 (read) and 2 (write) to get 6 (read, write).

The following table defines the numbers to obtain each mix of permissions:

```
User (Owner) Permissions
Group Permissions
Other Permissions
Modes

---------- 000  No permissions for any user
---x--x--x 111  Execute permission for all users
--w--w--w- 222  Write permission for all users
--wx-wx-wx 333  Write and execute permissions for all users
-r--r--r-- 444  Read permission for all users
-r-xr-xr-x 555  Read and execute permissions for all users
-rw-rw-rw- 666  Read and write permissions for all users
-rwxrwxrwx 777  Read, write, and execute permissions for all users
```

Thus to set the modes of a file to -rwxr-xr— you would type

```
chmod 754 file
```

Symbolic Modes The symbolic mode consists of three parts. They are:

who string	Who is affected (user, group or other)
operator	What actions are to be taken (add, remove, or absolute)
permissions	The permissions to add, remove, or absolute

The general format is:

```
chmod [ who_string ][ operator ][ permissions ] file_list
chmod [ who_string ][ operator ][ permissions ] directory_list
```

For example, the following command removes the write permission for all other users:

```
chmod o-w file
```

The o refers to other users, the - implies remove, and w is for write permission. No spaces are allowed in the symbolic mode strings.

Who string The who string controls which set of permissions are changed by the **chmod** command. Any combination of the following letters can be used for the who string:

u	The user (owner) of the file or directory
g	The users in the group that the file is associated
o	All other users on the system
a	All of the above (same as "ugo"). All user classes.
	If no who string is used, then the a is assumed. All user classes are affected.

Operators The operator controls what effect the permission strings will have on the permissions of the files or directories. The possible operators are:

–	Remove the specified permissions from the file or directory.
+	Add the specified permissions to the file or directory.
=	Assign the absolute permissions as specified. Permissions not specified are removed.

The = operator assigns the given permissions. It is the same as performing a **chmod 000 file** or a **chmod a-rwx file**, then performing the **chmod** to add the permissions specified after the = operator. For example, if you had a file with the following permissions:

```
rwxr—— 1  mylogin   ts      5 Feb  2 afile
```

and you typed

```
chmod a=rx
```

the file permissions would change to

```
r-xr-xr-x 1  mylogin   ts      5 Feb  2 afile
```

Permission String The permission string contains the permissions you want to add or remove from the current permissions. You can combine the following letters to create the permission string:

l	Mandatory file locking during file access
r	Read permission
s	Set-user-ID if used with the o who string or set-group-ID if used with the g who string
t	Sticky bit, hold text in memory until space is needed

w	Write permission
x	Execute permission
u	Indicate that *permission* is to be taken away from the current user
g	Indicate that *permission* is to be taken away from the current group
o	Indicate that *permission* is to be taken away from the current others

Multiple symbolic modes can be used on the same command line. Separate each set of symbolic modes with commas (,). For example,

```
chmod o+r,g-w myfile
```

would add read access for other users and remove write permission for group members.

More than one operator and permission sequence may be used for each who string. For instance,

```
chmod g+r-w myfile
```

would add read access for members of the group and remove write permissions.

You can remove all permissions by using the = operator and no permission string. For example,

```
chmod = myfile
```

removes all permissions for all users.

Special Modes The set-user-ID, set-group-ID, sticky bit, and file locking modes are special modes providing more flexibility in how UNIX handles the execution of a file.

The set-user-ID is reflected in the user execute position. The set-user-ID allows a different user to assume the user-ID of the file. Thus you execute the file (program) as if logged in as the owner of the file. For example, the following file belongs to the user "luwis":

```
-rwsr-xr-x    1  luwis        ts          1 Jan 20 17:11 testprog
```

If you execute the testprog file, you will execute it as though the user luwis was executing it. This is useful to provide access to secure data via controlled program environments.

The set-group-ID works on the same principal as the set_user_ID, except when you execute it, the group-ID of the file is assumed, not the user-ID. The set-group-ID is displayed as the group execute position as shown in the following example:

```
-rwxr-sr-x    1  luwis        ts          1 Jan 20 17:11 testprog
```

If either position is occupied by a capital S instead of a lowercase s, then you can access the file as though you are the owner but you cannot execute the file.

If the sticky bit is enabled, a lowercase t is placed in the other's execute position. The sticky bit informs the system to keep the program text of the file loaded in memory. For example, assume you execute the **testprog** program; when you exit from it the system will not unload the **testprog** from memory.

```
-rwxr-xr-t    1  luwis        ts          1 Jan 20 17:11 testprog
```

If a capital T is present, then the file is not executable and can never be loaded into memory. Therefore, it is an undefined permission or state of the file. Only the super-user can set the sticky bit.

File locking is enabled if the group execute position is set to lowercase l. The l represents mandatory file and record locking. If file locking is enabled, then a file is temporarily unavailable to all other users while it is being accessed. The following example shows the position of the l in the modes:

```
-rwxr-lr-x    1  luwis        ts        1 Jan 20 17:11 testprog
```

RELATED COMMANDS Refer to the **chgrp**, **chown**, and **ls** commands described in modules 16, 18, and 84, respectively.

APPLICATIONS

You can use the **chmod** command to allow or deny access to a file or directory. By removing read and write permissions for *other* users, you deny all users not in your group access to the file. If you removed the read and write permissions from the group permissions, then only you could access the data. In some cases you may want to remove your write permissions so you cannot accidentally remove or overwrite the file.

Using the **chmod** command to set the execute permissions allows you to execute a file that contains a shell script. You can also set the permission to l (lowercase L) for file locking. Then only one program or user can access the file at any given time.

The absolute mode is used to set all modes at once. It is generally used by programmers that like terse syntax. The symbolic mode is easier to understand and remember. It is also more useful for adding and removing only one or two specific modes.

TYPICAL OPERATION

In this activity you use the **chmod** command to change the modes of a directory. Begin at the shell prompt.

1. Type **chmod 700 letters** and press **Return** to change the permissions of the *letters* directory to read, write, and execute for you the owner only.
2. List the new permissions by typing **ls -ld letters** and pressing **Return**. Notice the dashes in the six right-hand positions of the permissions field. Only you have access to the letters directory, with exception to the super-user.

```
cj> ls -ld memos
drwx——   2 mylogin       ts        512 Feb 11 16:25 letters
```

3. Turn to Module 146 to continue the learning sequence.

Module 18
chown (SV)

DESCRIPTION

The external **chown** command lets you change the owner of a file or directory. As a normal user you can only change the ownership of files you already own. The super-user can change the ownership of any file, regardless of who owns the file. You can specify a user name or a user ID for the new owner.

BSD (Berkeley)

Some BSD systems allow the user to execute the **chown** command. But on most systems the **chown** command is a super-user-only type command.

COMMAND FORMAT Following is the general format of the **chown** command.

```
chown [ -hR ] owner file_list
chown [ -hR ] owner directory_list
```

Options The following options may be used to change how **chown** functions.

-h — Changes the owner of the symbolic links instead of the owner of the files pointed to by the symbolic links.

-R — Forces **chmod** to recursively descend the directory trees listed, changing the group ID of files and directories. If a symbolic link is encountered, its group ID is changed but it is not traversed.

Arguments The following arguments must be passed to the **chown** command.

owner — The name of a user or the decimal user ID number specifying the new owner.

file_list — One or more files to change the ownership from your user ID to the specified user ID of *owner*.

directory_list — One or more directories to change the ownership from your user ID to the specified user ID of *owner*.

FURTHER DISCUSSION You must specify a valid user name or user ID. Only the owner of a file or directory can change the ownership, except for root, who can change the ownership of any file or directory.

If the set-user-ID or set-group-ID mode is enabled, the **chmod** command will disable it unless you are the super-user.

POSIX NOTE If the operating system is configured with the {POSIX_CHOWN_RESTRICTED} option, then the owner of a file cannot change the owner of the file. Only root can change the owner ID in this configuration.

RELATED COMMANDS Refer to the **chgrp**, **chmod**, and **ls** commands described in modules 16, 17, and 84.

RELATED FILES The **chown** command checks the /etc/passwd file for the new owner or user ID given on the command line. If the user name or user ID is found and the file is changeable, then **chmod** performs the change.

APPLICATIONS

You may want someone else to own a file. If you are leaving the department or the company, you may want to give your files to other users. Since only the owner or the super-user can change the ownership, you may want to decide who needs which files. Thus you simply perform a **chown** for each file to change the ownership.

Typically, the super-user uses the **chmod** command more than any other user. Changing ownerships of files that have been changed by accident or left behind by someone for another user is a common reason.

TYPICAL OPERATION

In this activity you use the **chown** command to change the ownership of a temporary file, then you remove the file. Begin at the shell prompt.

1. First create a file called tempfile. Type **touch tempfile** and press **Return**. If you like, list the files owner using the ls -l command.
2. Now change the owner of the file by typing **chown nancy tempfile** and pressing **Return**. Now the file is owned by the user named nancy. You could have typed chown 51 tempfile to achieve the same results. This assumes there is a user named *nancy* whose user ID is 51.

NOTE

On some SV and all BSD systems this command will not work because you must be root to use chown. If your system returns an error like "not owner," skip to step 4.

3. Type **ls -l** and press **Return** to list the new owner of the tempfile.
4. Type **rm tempfile** and press **Return** to remove the tempfile. If you are asked for a confirmation, type **y** and press **Return**.
5. Turn to Module 16 to continue the learning sequence.

Module 19
clear

DESCRIPTION

The external **clear** command clears your terminal screen if possible. It searches your environment for the TERM(terminal) variable, then looks in the /usr/lib/terminfo/?/TERM database or the /etc/termcap file to determine how to clear the screen.

COMMAND FORMAT Following is the general format of the **clear** command.

```
clear
```

DIAGNOSTICS AND BUGS If your terminal variable is not set to the correct type of terminal, your screen may not be cleared. If your terminal does not clear and/or strange characters appear, you probably need to set your terminal type variable. Refer to Module 99 on how to set environment variables.

RELATED COMMANDS Refer to the **tput** command described in Module 139.

RELATED FILES The **clear** command looks up the proper characters to clear your screen in the /usr/lib/terminfo/?/TERM database or the /etc/termcap file.

APPLICATIONS

The **clear** command is used to clear your terminal screen. You may find this useful just to clean up your screen while working at your terminal. The **clear** command is often used in shell scripts to clear the screen before presenting a menu or data output. Some users place the clear command in their logout file, so when they log out their terminal screen is cleared.

TYPICAL OPERATION

In this activity you use the **clear** command to clear your terminal screen. Begin at the shell prompt.

1. Type **clear** and press **Return**. Your terminal screen should be cleared and your shell prompt will return at the top of the screen.
2. Turn to Module 139 (SV), Module 128 (BSD) to continue the learning sequence.

Module 20
cmp

DESCRIPTION

The external **cmp** command compares two files. If the files are the same, **cmp** returns with no output. If the files are different, **cmp** returns the line number and byte position of the first difference. Only two files may be compared at one time. The unique features of **cmp** over other file comparison tools are:

- Data can be of any type; does not have to be ASCII characters.
- Data does not have to be sorted.
- Execution is faster than most other compare programs.

COMMAND FORMAT Following is the general format of the **cmp** command.

```
cmp [ -ls ] - file2
cmp [ -ls ] file1 file2
```

The first format is used to compare the standard input to a file (file2). The second compares two existing files. You may find it helpful to use the first format in a pipe. The **cmp file1 -** format is not valid.

Options The following list describes the options that may be used to control how **cmp** functions.

-l	Display the byte position in decimal and the bytes that differ in octal for each occurrence of a difference.
-s	Do not display information for differing files. Only set the return code to 1.

Arguments The following arguments may be passed to **cmp**.

-	Read from the standard input for the first input files.
file1	The first of two files to compare. This may be a text or binary type file.
file2	The second of two files to compare. This file may also be text or binary.

FURTHER DISCUSSION If the - option is specified, you must enter input from your keyboard or pipe information to **cmp**. The -s option is useful if you only want to check the equality of two files. By checking the return code, using the $? ($status-csh) variable in the shell, you know whether the files are identical or different. The -l allows **cmp** to continue checking for differences throughout the file. By default, **cmp** only checks the files until the first difference is incurred, then it exits.

RELATED COMMANDS Refer to the **comm** and **diff** commands described in modules 22 and 33.

RETURN CODES The return codes of **cmp** may be used to control conditional statements within shell scripts. The following list describes the meaning of each code.

0 The files are identical.
1 The files are NOT identical.
2 There is a problem reading a file or with an option you provided.

APPLICATIONS

The **cmp** command is used to compare files for possible differences. You may want to think of it as the file comparison utility that can compare two files of any type for a difference of equality. It performs faster than the other file compare utilities. This is because it is a simple program comparing byte by byte. It can be thought of as a binary yes or no type file comparison program. Yes, the files are the same or no, the files are different. Thus it lends itself to shell scripts that need to check for files that match exactly, but they don't care what is wrong.

One example is the removal of like files. You can use the **cmp** command to compare files. If no output is returned (the return code is 0), you know the files are identical and thus can proceed to remove one of the files. On the other hand, if you want to make a backup copy of a file, you may want to use **cmp** to ensure the new copy is identical to the original. UNIX will usually complain if it did not make an identical copy. But if it adds security and bliss to your life, double-check the files!

Another example is trying to write a program that generates a predefined form. You may want to create the form in an ASCII file using an editor. Then generate the output of your program into a file and compare the new file with the predefined form using **cmp**. Once you have a matching blank form, you can continue writing your program to fill in the blank areas with variable output.

TYPICAL OPERATION

In this activity you use the **cmp** command to compare two files and check the status code. Begin at the shell prompt.

1. Type **cmp file1 file1** and press **Return**. Notice there should be no output on your screen. The two files are identical; they should be, they are the same file.
2. Now check the return code of **cmp** by typing **echo $?** ($status-csh) and pressing **Return**. The return code is 0 (zero) if no messages were returned to your screen.
3. Now compare two different files. Type **cmp /bin/ls /bin/cat** and press **Return**. Your screen should show a difference message such as the following one:

```
cj> cmp /bin/ls /bin/cat
/bin/ls /bin/cat differ: char 2, line 1
```

4. Check the return codes by typing **echo $?** ($status-csh) and pressing **Return**. Notice the return code is 1 instead of 0. This informs you that the two files are different.
5. Turn to Module 22 to continue the learning sequence.

Module 21
col

DESCRIPTION

The external **col** command removes control characters from the input text. This lets you display text on your terminal or send it to a printer that does not support such control characters. The input for **col** must always be the standard input (**col** cannot read from files). The output is always the standard output. The most common use of **col** is in a pipe command after an **nroff** type command. The **nroff** command is a typesetting processor. The **col** command accepts eleven different control code sequences from the input. The output is physically converted to appear as similar to the original input text as possible without using special characters.

COMMAND FORMAT Following is the general format of the **col** command.

```
col [ -bfpx ]
```

BSD (Berkeley)

```
col [ -bfh ]
```

Options The following options may be used to control how **col** functions.

-b Implies that the output device cannot backspace. Therefore, if multiple characters are to appear at the same location, only the last character is displayed. For example,

```
H<bs>_
```

displays _ (underscore); the H is ignored.

-f Does not remove half line feed (Esc-9) sequences from the text. Normally half line feeds are converted to full line feeds.

-p Passes unknown escape sequences to the output. Normally **col** removes all escape sequences from the text.

-x Suppresses the conversion of multiple space characters to tabs. By default, **col** converts a specific number of spaces to a single tab. Usually eight spaces are converted to one tab.

BSD (Berkeley)

-h Converts white space (spaces) to tabs to shorten printing time.

Arguments The **col** command does not accept filenames for input. Thus you must pipe the input to **col** or use indirection signs. For example,

```
cj> env | col
cj> cat mydoc | col
cj> col < mydoc
```

CONTROL CHARACTERS The control characters converted by **col** are listed in the following table. Each character or set of characters usually provides a common display control sequence.

Character	*Name*	*ASCII Code (OCTAL)*
Ctrl-H (^H)	Backspace (BS)	010
Ctrl-I (^I)	Tab (HT)	011
Ctrl-J (^J)	Linefeed (LF)	012
Ctrl-K (^K)	Vertical Tab (VT)	013
Ctrl-M (^M)	Carriage Return (CR)	015
Ctrl-N (^N)	Start-of-text (SO)	016
Ctrl-O (^O)	End-of-text (SI)	017
	Space (SP)	040

The following escape sequences are converted by **col**.

Esc-7	Reverse line feed	033 067
Esc-8	Reverse half line feed	033 070
Esc-9	Forward half line feed	033 071

DIAGNOSTICS AND BUGS The **col** command retains the last 128 lines of text. Therefore, it cannot back up more than 128 lines to perform certain reformatting.

The longest line must not exceed 800 characters, including control characters.

The first line of your document must not contain subscripting since **col** ignores control characters on the first line.

RELATED FILES The **col** command can only read from the standard input and write to the standard output.

APPLICATIONS

The most common use of the **col** command is with the text processing **nroff** command. But it is also handy for displaying files that contain control sequences that garble your terminal screen. For example, you may have a man page that you often reference stored in a file. If you created the man page on a different terminal type than the one you currently use, the man page may not appear correctly. Therefore, you might use **col** to remove the control characters so the man page appears in a readable format.

TYPICAL OPERATION

In this activity you use the **col** command to display a man page output. But your terminal type is set to the wrong type, so you would get garbled output on your screen if you did not use **col**. Begin at the shell prompt.

1. Type **echo $TERM** and press **Return** to display your terminal type.
2. Type **TERM=tvi925** and press **Return** to set your terminal to a different type. If you are using a tvi925 terminal, replace the *tvi925* value with *vt100*.

C Shell

If you are using the **csh**, type **setenv TERM tvi925** and press **Return**.

---

3. Type **man cat** and press **Return**. The man page should appear but not correctly.
4. Type **man cat | col** and press **Return** to display the man page so you can read it on your terminal.

NOTE

Normally you have the TERM variable set to the proper terminal type. This activity is just an example of how **col** functions.

5. Type **TERM=vt100** and press **Return** assuming that vt100 was your terminal type when you started this exercise.
6. Turn to Module 57 (BSD) to continue the learning sequence.

Module 22

comm -andate

DESCRIPTION

The external **comm** command compares two lists of sorted ASCII data (files). The default output is three columns; lines only in file1, lines only in file2, and lines in both files. Lines common to both lists or lines only in one of the two lists may be displayed. The two lists may be stored in files or one may be the standard input. The possible output displays can consist of:

- Lines only existing in the first file
- Lines only existing in the second file
- Lines existing in both files
- Any combination of the above

NOTE

Remember, when using **comm** the files must be sorted in ASCII collating sequence. Thus they must also be ASCII files.

COMMAND FORMAT Following is the general format of the **comm** command.

```
comm [ -123 ] - file2
comm [ -123 ] file1 file2
```

The first format is used to compare the standard input to a file (file2). The second compares two existing files. You may find it helpful to use the first format in a pipe.

Options The following list describes the options that may be used to control how **comm** functions.

-1	Suppress the displaying of column 1. Lines existing only in file1 are not displayed.
-2	Suppress the displaying of column 2. Lines existing only in file2 are not displayed.
-3	Suppress the displaying of column 3. Lines existing in both files are not displayed.

Arguments The following list describes the arguments that may be passed to the **comm** command.

-	Read from the standard input. The input is read from your keyboard or a pipe instead of a file.
file1	The first file to read input from and compare against file2.
file2	The second file to read input from and compare against file1.

FURTHER DISCUSSION Before you can use the **comm** command you must be sure that the input files are sorted. If they are not, **comm** will not function correctly. To sort files refer to Module 122.

You may find it useful to combine options to suppress multiple columns from displaying on your screen. For example, you may want to see only lines that exist in the second file. By using the following format,

```
cj> comm -13 file1 file2
```

you will find which lines of file2 do not exist in file1. This is useful for distinguishing which file has certain data in it.

RELATED COMMANDS Refer to the **cmp, diff, sort,** and **uniq** commands described in modules 20, 33, 122, and 149.

RETURN CODES The return codes of **comm** may be used to control conditional statements within shell scripts. The following list describes the meaning of each return code.

0 The files are identical.
1 The files were NOT identical.
2 There was a problem reading a file or with an option you provided.

APPLICATIONS

You can use the **comm** command to compare two lists of data. The two lists must reside in different files. You can compare for common items existing in both sets of data or for data that exists in one file and not the other.

One example would be to compare two lists of names. One file might contain members of a computer club who are single and a second file might contain members of the same club who work with UNIX. By using the command **comm -23 singles computers** you can list the singles who do not work with UNIX. Using the -12 option would list members who are single and work with UNIX. And the -13 option would list members who are not single and work with UNIX.

Since it can only function on ASCII files, it is limited to text type files.

TYPICAL OPERATION

In this activity you use the **comm** command to compare two files for like and dislike information. Begin at the shell prompt.

1. Type **cat > users** and press **Return** or use an editor to create the following file.
2. Type the following lines, press **Return** to end each line. Press **Ctrl-D** on the last line to end the input.

```
cj> cat > users
Barbara
David
George
Jill
Scott
Tina
^D
```

3. Type **cat > mgmt** and press **Return**.

4. Type the following lines, ending each line by pressing **Return**. Press **Ctrl-D** on the last line.

```
cj> cat > mgmt
Andrew
Barbara
Kathy
Scott
^D
```

5. Type **comm users mgmt** and press **Return** to display the three-column output. Column one contains lines only in the *users* file. Column two contains lines only in the *mgmt* file. Column three has lines contained in both files.

```
cj> comm users mgmt
        Andrew
                Barbara
David
George
Jill
        Kathy
                Scott
Tina
```

6. Type **cat users | comm -12 - mgmt** and press **Return** to display who is in both files. The - represents the standard input and thus allows **comm** to read from the pipe.

```
cj> cat user | comm -12 - mgmt
                Barbara
                Scott
```

7. Turn to Module 124 to continue the learning sequence.

and ate

d eat h

& → S

↓

english alphabet 'S'

Module 23
COMMANDS

DESCRIPTION

This module discusses the different aspects of how UNIX and the shell handle commands. There are five areas of discussion:

- Simple Command Syntax
- Shell Command Syntax
- Command Execution
- Internal and External Commands
- Return Codes

SIMPLE COMMAND SYNTAX This section describes the syntax used to invoke a single UNIX program or utility, i.e., simple command. The following example explains the syntax in detail.

```
name [ -option ... ] [ cmdarg ... ]
```

where:

[]	Brackets enclose an *option* or *cmdarg* that is optional.	
...	Periods indicate multiple *options* or *cmdargs* may follow.	
name	Is the filename or full path of an executable command.	
option	Preceded by a "-" (hyphen). Must be one of the following.	
	noargletter ...	A single-letter option. Multiple *noargletters* can be grouped after one "-". Perhaps explaining the noargletter word would help. The *noarg* part means no arguments will follow the *letter*.
	argletter optarg[,...]	A single-letter option requiring an option argument. This is to say an argument will follow the option. Bascially, the option informs the program how to use the following argument.
	optarg	An optarg is a character string that is an argument to the previous *argletter* option. See rule 8 for further syntax rules.

BSD (Berkeley)

Command options in BSD may have to be separated by spaces. For example, on System V you might have -aik, whereas on BSD you would have to specify -a -i -k. Unfortunately, BSD has not implemented a standard on command line formats. Check the INTRO section of your system User's Reference Manual for command definitions.

cmdarg A character string not beginning with a "-". Often a filename path. A "-" by itself represents the standard input.

| A vertical bar may be placed between parameters to create a conditional "or" selection of parameters. Meaning either the preceding parameter or the following parameter may be used but not both.

The long form of a command is written as follows:

```
name [ -noargletter ... ] [ -argletter optarg ... ] [ cmdarg ... ]
```

The following example is described to help you understand how the shell interprets a command.

```
date [ mmddhhmm[yy] ] | [ +format ]
```

The command name is **date**. The syntax shows that all parameters are optional. It is not necessary to provide any options. If the first argument is used, then yy is optional. The vertical bar represents an "or" condition in the parameter list. Therefore, either the first parameter or the second one can be used but not both at the same time.

Another command might be worth looking at:

```
cat [ - ] [ -su ] [ -v [ -et ] ] [ file_list ]
```

The command name is **cat**. All parameters are optional. The **-** must be surrounded by *blanks*. The **-v** must be specified if either the **-e** or **-t** option is specified. The **s**, **u**, **v**, **e**, and **t** options may all be placed after the same hyphen (-). The command syntax is intended to show the dependencies of arguments and options, which is not necessarily the final format.

Command Syntax Rules The following list of rules will be enforced on all releases of UNIX System V from Release 3.0 and later; although, there are some commands that existed before the 3.0 release that do not conform to these exact rules. The **getopts** command provides command parsing for shell procedures. It will check for legal options. It supports all but rules 1 and 2, which must be enforced by the command name itself.

1. A command *name* must be in the range of two to nine characters long.
2. A command *name* must contain only lowercase characters and numbers.
3. Each *option* (noargletter and argletter) must be only one character long.
4. Each *option* or group of *options* must be preceded by a "-".
5. All *Noargletter* type options may be grouped together after one "-".
6. The *optarg* list must be preceded by at least one *blank* or *white space*.
7. An *optarg* cannot be optional. If an *argletter* is specified, then an *optarg* must follow.

8. Multiple *optargs* must be separated by commas or must be separated by white spaces and quoted (e.g., -o abc,def or -o "abc def").
9. *Options* must precede *cmdargs*.
10. A "--" string may be used to indicate the end of the *options*.
11. The relative order of the *options* should be irrelevant.
12. The relative order of the *cmdargs* may affect how the command functions. The order may be significant.
13. The " - " (a hyphen preceded and followed by blanks) should only be used to indicate the standard input.

SHELL COMMAND SYNTAX The shell's sophisticated programming language allows complete control of command execution and combination of commands. The shell supports multiple types of commands providing for a flexible and powerful command and programming language.

Simple Commands A simple command is a list of words. The general format of a simple command follows. For a complete description refer to the Simple Command Syntax section of this module.

```
command options arguments
```

The first word is the name of the program to be executed. All remaining words are passed to the program as positional arguments. The first word equates to positional parameter 0. The following example illustrates this format.

```
cj> cat -s file
```

cat is positional parameter 0, -s is positional parameter 1, and file is positional parameter 2.

Keyword Commands A keyword command is a shell programming construct. The command must begin with one of the following unquoted words.

```
if then else elif fi case esac for while until do done function { }
```

Keywords must be preceded and followed by a blank. For example,

```
{echo A FILE ; cat /etc/group}
```

results in two errors. The first is "{echo" is not a valid command. The second is that "/etc/group}" is not a valid file. But the command,

```
{ echo A FILE ; cat /etc/group }
```

works properly because the braces are interpreted correctly as keyword commands.

Pipeline A pipeline is one or more simple or keyword commands connected by a vertical bar (|). Each command is executed as a separate process. In a pipeline the standard output of the previous command is "piped" into the standard input of the following command. The shell waits for the last command in the pipeline to complete. The exit status of a pipeline is the return code of the last command in the pipeline. The general format of the pipeline follows.

```
command | command
```

For example,

```
nawk -F: '{print $1}' /etc/group | sort | pr -t4
```

generates a list of valid groups sorted in four-column output. The standard output of **nawk** is piped into the standard input of **sort**. Then the standard output of the **sort** command is read by the standard input of **pr**. The **pr** command does not format columns on BSD. Sorry!

List A list is one or more pipelines separated by ;, &, && or ||. A list may end with a semicolon (;) or an ampersand (&). The pipe symbol (|) has precedence over these four operators. The ; and & have equal precedence and higher precedence than the && and || operators. The && and || operators have equal precedence. The general format of a list command follows

```
pipeline ; pipeline

pipeline & pipeline

pipeline || pipeline

pipeline && pipeline
```

where pipeline is a combination of simple and keyword commands.

To simplify discussion on commands, the word command is used to describe all of the simple, keyword, pipelines, and lists discussed in this book.

COMMAND EXECUTION When you press Return to send a command to UNIX, the shell parses the command and performs the required substitutions and expansions. The following list outlines the sequence of substitutions and expansions in the order in which they are performed.

- Aliasing (Module 4)
- Tilde Substitution (Module 14 or 71)
- Command Substitution (Module 71)
- Variable Substitution (Module 99)
- Blank Interpretation (Module 71)
- File Name Generation (Module 54)
- Arithmetic Evaluation (Module 73)

Once the command has been expanded it is executed. If an internal or keyword command is used, the shell executes the command within the current environment (no subshell is spawned). User defined functions are included in the keyword commands. If I/O redirection is performed, a subshell is created and the commands are executed in the new environment.

If the command is an *external* type, then a process is always created and the shell attempts to execute the command via the **exec** system call. Internal and external commands are described in a following section of this module.

Searching for Commands The shell interprets the first parameter on the command line ($0) and, based on the following criteria, locates the program to be executed.

If the command contains a / then the shell searches only the directory specified. The directories in the PATH variable are not searched for the command. If the command is not in the directory, the shell returns a "not found" message. For example,

```
cj> /usr/bin/lx
ksh:/usr/bin/lx: not found
cj> ../bin/lx
ksh:../bin/lx: not found
```

If you do not specify a directory path in the command name (no / is used), the shell searches each directory listed in the PATH variable for the command. If the command is located in one of the directories and is executable, then it is executed. If it is not found in one of the directories, the shell notifies you as follows:

```
cj> lx
ksh: lx: not found
```

Tracked Aliases By using **alias** or **set** you can have the shell remember where a command is for later use. If a command has been tracked, the next time you attempt to execute the command the shell will not have to search the PATH directories for the command. This reduces overhead and response time in command searching.

INTERNAL AND EXTERNAL COMMANDS There are two basic types of commands: the external and internal commands.

External Commands An external command is a computer program. It is a compiled binary file. The shell searches the directories in the PATH variable to find external commands, or the full path is specified on the command line. Once the command has been located, the shell must spawn a child process for the program. Then the shell attempts to execute the program by making an exec system call. If successful, the program is loaded into memory and the UNIX operating system executes the program as a new process.

Internal Commands An internal command is built-in to the shell code. It is a function or subroutine compiled as part of the shell program (**ksh**, **sh**, **csh**). The shell does NOT have to search the PATH variable for an internal command. No new process is spawned to execute the command. Thus no disk reading or program loading is required. The result is much faster execution.

So why not put all commands in the shell? There are multiple arguments concerning this idea. The conventional operating system concept actually leans more towards command processors and standard commands all being a part of the operating system. But UNIX tends to be more modular oriented. The modularity provides for more flexibility and better management. The more commands that are placed in the **ksh** program, the larger and less responsive it becomes. Thus the argument is fast response up to a break point versus management of programs.

RETURN CODES Return codes are two bytes of status information returned to the shell after each command terminates. The first byte is returned by the system. It provides the cause for termination. If normal system termination occurred, a zero is returned. The second byte is returned by the program or command. Upon successful termination the default return code is zero (0). If the program detects problems and terminates, then a nonzero return code is placed in the second byte. Return codes may be referred to as "exit code," "exit status," "return code," or "return status."

The return code is assumed to be zero if the program terminated under normal conditions. Therefore, some of the modules in this book do not have RETURN CODE sections, because the return codes are zero and nonzero with no special meaning besides normal and abnormal termination.

The shell assigns the return code to the ? variable. You can refer to the variable by using ${?} ($status-csh) or $?. The return code is often used in shell scripts to check whether a program terminated correctly or incorrectly.

There are several ways to test return codes. The following code checks for the existence of positional parameter 1 in the passwd file as a login name.

```
cj> cat valid_user
if grep "^$1" /etc/passwd > /dev/null
then echo "$1 is a valid user." ; exit 0
else echo "$1 is NOT a valid user."; exit 1
fi
```

The return code of the **grep** command is tested by the if command. If the return code is 0 the command is true, else it is false. The **grep** command searched the /etc/passwd file for a string passed to the shell script as positional parameter 1. If the string matches a string at the beginning of a line (^) in the passwd file, the line is displayed. The indirection of all standard output from the **grep** command is sent to the system's trash can (/dev/null), so you do not see the output from the command. If you ran this as the command valid_user, the return code would be 0 if the user was found, and 1 if not found.

APPLICATIONS

Understanding commands is an essential requirement before you can use the UNIX system. Commands are the fundamental part of the interface language that allows you to perform tasks on the computer. The UNIX commands are unlike most commands offered by an operating system. They can be used to perform hundreds of programming tasks to solve everyday problems. It is advisable to become familiar with command syntax, rules, and execution. If you do write shell scripts, be sure to follow the rules outlined in this module. It makes life easier for you and the users in the long run.

TYPICAL OPERATION

In this activity you use the shell to execute internal and external commands. Begin at the shell prompt.

1. Type **cat - file1 > file2** and press **Return**.
2. Type the following text; end each line with **Return**. Press **Ctrl-D** on the last line to send **cat** an end-of-file signal. The text you typed is redirected to file2. When you press Ctrl-D, **cat** begins reading file1. Your input and file1 are redirected to file2. The **cat** command is an external command.

   ```
   cj> cat - file1 > file2
   This input is from my keyboard.
   Cat reads it from the keyboard and the shell redirects it.
   ^D
   ```

3. Type **cat file2** to display the contents of file2. Notice the input you typed from the keyboard and the contents of file1 are displayed.
4. Type **set -- one two three** and press **Return**. This sets the positional parameters of your login shell to "one," "two," and "three." The **set** command is an internal command.

C Shell

Type **set argv=(one two three)**

---

5. Type **echo $* " : " $2** and press **Return** to display the new positional parameters that you have set by using the set command. A colon followed by the second positional parameter is also displayed. The **echo** is an internal and external command. If the output is redirected, it is executed as an external command; otherwise, it is executed as an internal command.

C Shell

Type **echo $*":"$2** or **echo $argv":"$argv[2]**.

---

6. Turn to Module 67 to continue the learning sequence.

Module 24
cp

DESCRIPTION

The external **cp** command copies a file. It reads the contents of a file and creates a new file or overwrites an existing file. There are two basic formats of the **cp** command that allow you to:

- Copy one file to another
- Copy mutliple files to a directory
- Copy input from your keyboard to a file
- Copy a file to your terminal

COMMAND FORMAT Following is the general format of the **cp** command.

```
cp    [ -ip ]  source_file destination_file
cp    [ -ipr ] source_file_list destination_directory
cp -r [ -ip ]  source_directory destination_directory
```

Options The following list describes the options that may be used to control how **cp** functions.

-i	Interactive confirmation is required. You are prompted if the copy will overwrite an existing file. If your answer contains a **y**, the copy is performed. Otherwise, the copy is not performed.
-p	Preserve the characteristics of the source_file. Copy the contents, modification times, and permission modes of the source_file to the destination files.
-r	Recursively copy any source directories. If a directory is given as the source file, then all of its files and subdirectories are copied. The destination must be a directory.

Arguments The following list describes the arguments that may be passed to the **cp** command.

source_file	The existing file that will be copied
source_file_list	The name of files and/or directories to be copied to a new destination directory
destination_file	The name of the file the new copy will be named
destination_directory	The name of the directory where a copy of the files will be made
source_directory	The name of the directory from where copies are read

FURTHER DISCUSSION The first format copies one ordinary file to a new file. The source_file is the pathname of the file being copied. The destination_file is the pathname of the file being copied to by **cp**.

The second format will copy one or more files to a specific directory. The source_file_list is a list of pathnames of the files being copied. If you specify a directory with the -r option, the entire directory tree is copied to the destination directory.

The third format allows you to copy an entire directory structure to a new directory.

If the destination_file exists and you have write permissions, **cp** will overwrite the contents of the file. The access permissions and ownership of the overwritten file will be used for the new destination_file.

If the destination_file does not exist, the access permissions of the source_file are used. Your ownership and group IDs are used for the new file if you are the user performing the **cp** command.

You cannot copy a file to itself.

Some common formats are:

cp letter letter.bak	# make a backup copy of letter
cp letter ../tmp	# copy letter to temporary working directory
cp letter ../LTRS	# copy letter to original location (mv is better)
cp /dev/tty note	# copy input from keyboard to file "note"

RELATED COMMANDS Refer to the **ln, mv, rm, chmod, chown,** and **ls** commands described in modules 75, 91, 114, 17, 18, and 84.

APPLICATIONS

The **cp** command is used to make a copy of an existing file. It provides a way to create backup copies of files. If you accidently erase the current copy you are working on, you can fall back to the last backup copy you made.

You can copy a file to a temporary work area ($HOME/tmp is a good place) to perform changes. Then when you finish with the updates, copy the file back to its orginal location. You may want to write a shell script to perform these few tasks.

You may also want to use it to back up an entire work directory to a backup directory. Even though this uses up disk space, it may be well worth the cost. The trade off depends on the importance of the work you are performing versus the time it would take to recover or recreate the files if they are lost.

TYPICAL OPERATION

In this activity you use the **cp** command to copy a file. Begin at the shell prompt.

1. List your HOME directory by typing **ls** and pressing **Return**. Notice the file named *file1* in the following display.

```
cj> ls
bin     calendar  db   file1     letters   stuff
```

2. Copy *file1* to *stuff* by typing **cp file1 stuff** and pressing **Return**. This overwrites the contents already contained in the file named *stuff*.
3. Type **cp calendar letters** and press **Return** to copy the calendar file to the letters directory.
4. Turn to Module 114 to continue the learning sequence.

Module 25
cpio (SV)

DESCRIPTION

The external **cpio** command is used to copy files and directory structures in and out of archive files. It can also copy directory structures from one location to another. There are three modes of operation handled by **cpio**. The "copy out" mode creates an archive file. The "copy in" mode allows you to retrieve a previously archived file. The "pass" mode passes a copy of a directory structure from the source to a new destination. The new directory structure is an exact image of the source structure.

COMMAND FORMAT Following is the general format of the **cpio** command.

```
cpio -i[ /6bBcdfkmrsStuvV/ ] [ -C size ] [ -E file ] [ -H hdr ] [ -O file ]
[ -I file [ -M msg ]] [ -R ID ] [ patterns ]
cpio -o[ /aABcLvV/ ] [ -C size ] [ -H hdr ] [ -O file ] [ -M msg ]
cpio -p[ /adlmruvV/ ] [ -R ID ] directory
```

The first format is the "copy in" mode, the second is the "copy out," and the third is the "pass" mode.

NOTE

Not all systems support the -r option in PASS mode.

NOTE

The **cpio** command is not a standard utility on BSD. But many vendors include it in their release of BSD. It is a standard SV utility.

Modes The **cpio** command must have one of the three options -i, -o, or -p specified on the command line. Each of these options places **cpio** in one of three modes. The following list describes each of these modes.

COPY IN

-i *patterns* Causes **cpio** to extract files from the standard input. This usually implies input redirection on the command line. It assumes the input is the result of a previous **cpio -o** command. For example,

cpio -i < cpio.i

assumes you have previously created the file *cpio.i* using **cpio** and attempts to extract all files archived.

If you do not specify any *patterns*, all files in the archive file are extracted. If you do specify *patterns*, only the files that match the *patterns* are extracted. The *patterns* you use may contain any of the shell's Filename Generation Characters (?, *, [*class*]). You may specify multiple *patterns*.

The permissions of the extracted files are the same as the permissions when the files were stored in the archive file. The owner and group IDs of the extracted files are set to your owner and group ID unless you are super-user. In which case the owner and group IDs at the time of archival are restored.

NOTE

If a file exists on disk and **cpio** tries to extract a file by the same name, the extracted file will replace the disk file only if it (the extracted file) is newer. If the disk file is the same age or newer, **cpio** displays a warning message but does not overwrite the disk file.

CAUTION

When extracting files be aware of how the files were copied out. It is not uncommon for relative pathnames to be used. Thus when you extract a file it may actually be placed in your parent directory instead of in the current directory or some other directory. For example, if a **cpio** archive file was made using the following command,

```
ls .. ../.. | cpio -o > arch
```

the files stored in the *arch* file would have pathnames of ../file1, ../../fileX, etc. When you extract the files, they would be placed in the parent and parent's parent directory. To check the pathnames of files in an archive file use the -t option.

COPY OUT

-o

The copy out option causes **cpio** to read the standard input to acquire a list of pathnames (files and directories). It then copies each of the specified files to the standard output. The pathnames and header (inode) information is copied in conjunction with each file. For example,

```
ls | cpio -o > arch
```

copies all of the files in the current directory to the *arch* file.

CAUTION

When creating an archive file using **cpio** you should not begin pathnames with a /. Instead use relative pathnames. Either start with a . or the name of a subdirectory. This allows you to extract a file or the entire archive file to a new location. By doing this you can restore files from an archive file without overlaying the current copy of the file.

PASS

-p *directory*

The pass option allows you to copy a directory tree from a source location to a new destination. It works somewhat like the copy out option except the output is written to a *directory*. The files in the current directory are copied to the new directory. For example,

```
ls | cpio -p /u2/doc/aba
```

copies all the files in the current directory to the directory /u2/doc/aba.

Options The following options may be used to control how **cpio** functions.

-6 Causes **cpio** to process an old UNIX Version 6 format **cpio** archive file. Only usable with the -i option.

-a Causes the access times of the input files to be reset to the time they had before **cpio** archived them. Without the -a option the access times are updated. Access times for linked files are not reset.

-A Appends files to a floppy disk or hard disk archive. Requires the -O option.

-b Reverses the order of the bytes in a word. A word is four bytes long. For example, if a word contained "1234," the result would be "4321." Only usable with the -i option.

-B Changes the input and output block size to 5120 bytes. This speeds up I/O in certain cases. This is only useful on character special devices, such as tape drives, for example, a 9-track tape drive /dev/rmt/0m (/dev/rmt16 - BSD).

-c Generates internal header information in ASCII character format. This provides for better portability among different computers. If you plan on creating a tape on one system and loading it on another, you should use this option.

-C *size* Specifies Input/Output buffers to be *size* byte records. The default buf*size* is 512 when -C or -B are not used. Only effective when I/O is from/to a character special device and not a disk.

-d Forces **cpio** to create directories as needed. This is used in conjunction with the -i and -p options to create directories that exist in the archive file or in the source directory structure.

-E *file* Specify a file *file* which contains a list of filenames to extract from the archive, one filename per line.

-f Reverses the selection of *patterns*. All files that do not match the given *patterns* are copied in from the archive file.

-H *hdr* Read or write header information in *hdr* format. Mutually exclusive with the -c and -6 options. Use this option when the originating and destination systems are different operating systems and/or architectures. The valid *hdr* values are

crc CRC	ASCII header with expanded device numbers and per-file checksums.
ustar USTAR	IEEE/P1003 Data Interchange Standard header and format.
tar TAR	Tar(tar) header and format.
odc	ASCII header with unexpanded(small) device numbers.

-I *file* Read contents of *file* as the input archive. If *file* is a character special device, when the current medium(floppy diskette) is completly read, replace it and press Return to read the next one. Only usable with the -i option. See -M option.

-k Attempt to skip corrupted file headers and I/O errors. Normally cpio terminates abnormally when errors are encountered.

-l If possible, link files instead of copying them. Only usable with the -p option.

-L Follow symbolic links.

-m Retains the modification time of the file from the time of archival. Modification times of directories are not retained; only times of files are kept.

-M *msg* Defines a message *msg* to use when the -I or -O options are specified with a character special device. The message is displayed when the end of the media is reached. A single %d may be used in the message to display a sequence number of the next medium needed.

-O *file* Writes the output to file *file*. If *file* is a character special device and is full, replace the current medium and press Return to continue. See -M option.

-r Allows interactive renaming of files. If you type in a Return after a prompt with no filename, the file is skipped. This option cannot be used with the -p option.

-R *ID* Reassigns ownership and group permission for each file to user ID *ID*. Only the super-user can use this option. The user ID must be a valid ID in /etc/passwd.

-s Swaps the bytes within each halfword. A word is made up of four bytes. This is the same as swapping every other byte. Thus it provides a way to swap the high and low order bytes on a two-word machine.

-S Swaps every halfword with the next halfword. Swaps every two bytes with the next two bytes. You can only use this option with the -i option.

-t Displays a table of contents for the files stored in the archive file. No files are extracted or created.

-u Unconditionally copies files to their destination. Normally an older archive file with the same filename as an existing file will not overwrite the existing file.

-v Causes **cpio** to be verbose in its actions. Displays a list of filenames being archived or retrieved. If you use -v with the -t option, the output resembles the output of the **ls -l** command.

-V Terse verbose. A dot is printed for each file read or written. Assures you that **cpio** is working without displaying the filenames.

Arguments The following list describes the arguments that may be passed to the **cpio** command.

patterns Patterns are used in conjunction with the -i option to control file retrieval. You can use the shell's filename generation characters to form regular expressions to match files you want to retrieve from an archive file.

directory The pathname of the destination directory for the -p option.

FURTHER DISCUSSION If you are copying in or out on a character special device and **cpio** reaches the end of the medium, it will prompt for a new device pathname. For example, if you are saving to a floppy diskette and the diskette becomes full, **cpio** will display the message,

```
cj> find . -print | cpio -ocB > /dev/rdiskette
If you want to go on, type device/file name when ready.
```

At this point replace the floppy diskette with an empty one and type in the device pathname for the floppy drive (/dev/rdiskette) and press Return. If you do not want to continue the **cpio** process, you can just press Return and **cpio** will exit.

Copying/Extracting Files The **cpio** command can be used to copy files to an archive file. You can use various ways to provide the input control pathnames to **cpio**. However you choose, the input format has to be the same. The **cpio** command looks for pathnames to files you wish to have archived. For example,

```
./myfile
./file1
./file2
```

would save these three files from the current directory.

NOTE

The input list of pathnames for **cpio** must contain only one pathname per line. For example, the output of **echo** is not valid, nor is the output of **ls -x**.

To generate a list of pathnames, you can use the **cat**, **find**, **ls**, or any other command that produces a single-column list of pathnames. Some examples follow.

```
cj> ls | cpio -oc > /bck/032689
```

This sends **cpio** a list of the current directory to be saved in the archive file named 032689 in the /bck directory.

To extract the files, simply send the archive file to **cpio**'s standard input. This can be done in various ways. The following two examples accomplish the same task.

```
cj> cat /bck/032689 | cpio -ic
```

or

```
cj> cpio -ic < /bck/032689
```

Both of these commands restore all of the files archived in the /bck/032689 file.

Copying Directory Structures (Trees) Creating archives of directory structures works almost the same way as for files. The difference is you have to provide full pathnames for all files you wish to have archived. So not only is a list of files provided, but also all subdirectories you desire to save must be sent to **cpio**. The most common way to accomplish this task is with the **find** command. For example,

```
find . -depth -print | cpio -o > /dev/rdiskette
```

copies all of the files in the current directory and all subdirectories and files to the floppy diskette loaded in the /dev/rdiskette disk drive. Extracting files from this type backup works like normal file retrieval, with the exception you add the -d option so directories are created as they are needed.

RELATED COMMANDS Refer to the **cp**, **find**, **ls** and **tar** commands described in modules 24, 55, 84, and 133.

RELATED FILES The **cpio** command reads control information from the standard input to perform archival and retrieval functions. It writes the archive file to standard output. It writes retrievals to the proper location in the file system (directories).

BUGS/NOTES There are several incompatiblities and quirks you should know about cpio. They are as follows.

An archive created with the -c option on SV Release 4.0 cannot be read on SV Release 3.2 or earlier systems. The -H odc header in Release 4.0 is equivalent to -c headers on earlier SV releases.

SV Release 3.2 and earlier do not understand symbolic links. Thus restoring a symbolic link to an earlier SV release causes an actual file to be created.

Pathnames are restricted to 1024 bytes unless the -H odc option is used. In which case they are restricted to 256 bytes.

Only the super-user can copy special files (devices).

Blocks are reported in 512-byte increments.

If a file has 000 permissions, contains 1 or more characters, and the user is not root, the file will not be saved or restored.

APPLICATIONS

You can use the **cpio** command to create an archive file that contains files or entire directory structures. You can later use these archive files to rebuild or retrieve any files stored in the archive file. The super-user may use cpio to create an entire backup of a file system. This is done by using the **find** command with the **cpio** command. If you release products, you may find it useful to archive a subdirectory or multiple directory structures.

It can also be used to copy an existing directory tree from one location to another. This allows the super-user to easily move users from one file system to another, alleviating disk space problems and providing better utilization of the system's disk space.

Normally, **cpio** is used to transport data between systems. It writes data in a stream format, thus saving space and time when creating a tape backup. For archival of information to be used by the program linker (**ld**) you use **ar**. If you plan on saving archives to tape for backup purposes, you might want to consider using the **tar** command.

TYPICAL OPERATION

In this activity you use the **cpio** command to store a set of files, list the table of contents of the archive file, and unarchive a few of the files. Begin at the shell prompt.

1. Type **ls | cpio -oc > /tmp/mycpio** and press **Return**. This creates an archive file containing all the files in your current directory.
2. Type **cpio -ict < /tmp/mycpio** and press **Return** to display a table of contents for the files stored in the archive file.
3. Type **cpio -icr < /tmp/mycpio** and press **Return** to rename some of the files as they are restored from the archive file.
4. Press **Return** on each file until you reach the file named *file1*.
5. When cpio prompts with the file named *file1*, type **myfileagain** and press **Return**.
6. Continue pressing **Return** until **cpio** exits. If you like, you can press **Del** to exit **cpio** immediately. You now have a new file named myfileagain.
7. Turn to Module 133 to continue the learning sequence.

Module 26
crontab (SV)

DESCRIPTION

Use the external **crontab** command to create a file containing control statements to be interpreted by the **cron** command. Each statement consists of a time pattern and a command. The **cron** program reads your crontab file and executes the commands at the time specified in the time patterns. The commands are usually executed by a Bourne shell (sh).

COMMAND FORMAT Following is the general format of the **crontab** command.

```
crontab [ file ]
crontab -e [ username ]
crontab -l [ username ]
crontab -r [ username ]
```

Options The following options may be used to control how **crontab** functions.

-e Edit your crontab file using the editor defined by the EDITOR variable.

-r Removes your current crontab file. If username is specified then remove that user's crontab file. Only root can remove other users' crontab files.

-l List the contents of your current crontab file.

Arguments The following list describes the arguments that may be passed to the **crontab** command.

file The name of the *file* you want to be used as your crontab file. The *file* is copied to a file named *username* in the system crontab directory.

If you do not specify a filename for crontab to read as input, crontab reads the standard input until you press Ctrl-D.

FURTHER DISCUSSION The **crontab** command reads a file or the standard input to a directory that contains all users' crontab files. You can use **crontab** to remove your crontab file or display it. You cannot access other users' crontab files in the crontab directory.

Crontab File Format The crontab file contains lines that consist of six fields separated by blanks (tabs or spaces). The first five fields are integers that specify the time the command is to be executed by cron. The following table defines the ranges and meanings of the first five fields.

Field	*Range*	*Meaning*
1	0-59	Minutes
2	0-23	Hours (Midnight is 0, 11 P.M. is 23)
3	1-31	Day of Month
4	1-12	Month of the Year
5	0-6	Day of the Week (Sunday is 0, Saturday is 6)

Each field can contain an integer, a range, a list, or an asterisk (*). The integers specify exact times. The ranges specify a range of times. A list consists of integers and ranges. The asterisk (*) indicates all legal values (all possible times).

The following examples illustrate the format of typical **crontab** time patterns.

Time Pattern	*Description*
0 0 * * 5	Run the command only on Thursday at midnight.
0 6 1,15 * 1	Run the command at 6 a.m. on the first and fifteenth of each month and every Monday.
00,30 7-20 * * *	Run the command every 30 minutes from 7 a.m. to 8 p.m. every day.

NOTE

The day of the week and day of the month fields are interpreted separately if both are defined. To specify days to run by only one field, the other field must be set to an asterisk (*). In this case the asterisk means that no times are specified.

The sixth field contains the command that is executed by **cron** at the specified times. The command string is terminated by a new-line or a percent sign (%). Any text following the percent sign is sent to the command as standard input. The percent sign can be escaped by preceding it with a backslash (\%).

A line beginning with a # sign is a comment.

Each command in a crontab file is executed via the shell. The shell is invoked from your HOME directory (defined by $HOME variable). If you wish to have your (dot) .profile executed, you must specify so in the crontab file. For example,

```
0 0 * * 1   . ./.profile ; databaseclnup
```

would cause the shell started by **cron** to execute your .profile, then execute the program *databaseclnup*. If you do not have your own .profile executed to set up your environment, **cron** supplies a default environment. Your HOME, LOGNAME, SHELL, and PATH variables are set. The HOME and LOGNAME are set appropriately for your login. SHELL is set to /bin/sh and PATH is set to :/bin:/usr/bin:/usr/lbin.

NOTE

Remember not to have any **read** commands in your .profile which prompt for input. This causes problems when the cron job executes.

Command Output If you do not redirect the standard output and standard error of a command executed from your crontab file, the output is mailed to you.

Access To use the **crontab** command you must have access permission. Your system administrator can make the **crontab** command available to all users, specific users, or no users. Two files are used to control who can and cannot access the command. The /usr/sbin/cron.d/cron.allow file contains a list of all users who are allowed to use **crontab**. The /usr/sbin/cron.d/cron.deny file contains a list of all users who are denied access to **crontab**. If the cron.allow file exists but is empty, then all users can use the **crontab** command. If neither file exists, then no users other than the super-user can use **crontab**.

Displaying your crontab If you have a crontab file in the system crontab area, you can list it by typing **crontab -l**. If you do not have a crontab file, **crontab** returns the following message:

```
crontab: can't open your crontab file.
```

DIAGNOSTICS AND BUGS The **crontab** command will complain about various syntax errors and time patterns not being in the valid range.

CAUTION

If you type **crontab** and press **Return** without a filename, the standard input is read as the new crontab entries. Therefore, if you inadvertently enter crontab this way and you want to exit without destroying the contents of your current crontab file, press the Del key. Do not press the Ctrl-D key; if you do, your crontab file will only contain what you have currently typed.

RELATED FILES The **crontab** command uses the following files to execute your jobs as required.

/usr/sbin/cron.d	The main directory for the cron process.
/usr/sbin/cron.d/log	Accounting information for cron processing.
/usr/sbin/cron.d/crontab.allow	A file containing a list of users allowed to use crontab.
/usr/sbin/cron.d/crontab.deny	A file containing a list of users not allowed to use crontab.
/usr/spool/cron/crontabs	Location of crontab text to be executed.

APPLICATIONS

You can use the crontab file to create a list of commands that are run every minute, hour, day, week, month, or year. You can perform nightly database queries or weekly database reorganizations using a crontab entry. Reports can be created and printed every morning automatically. Any task that is done on a regular basis is a candidate for crontab.

The most common use of the **crontab** command and **cron** facilities is system administration chores. File systems can be checked, database maintenance, accounting, and various other tasks can be performed by using **crontab** to create a table containing the necessary commands.

TYPICAL OPERATION

In this activity you use the **crontab** command to create a crontab file and then remove the crontab file. Begin **crontab** at the shell prompt.

1. Type **crontab** and press **Return**.

2. Type in the following line; press **Return** at the end of the line. Then press **Ctrl-D** at the beginning of the next line.

```
cj> crontab
0,5,10,15,20,25,30,35,40,45,50,55 * * * * echo "This is a test!"
^D
```

WARNING

Commands will be executed with /bin/sh.

If you are allowed to have a crontab, the command will not complain and your shell prompt returns. If you are not allowed to have a crontab file, crontab will display a message informing you of the condition. Contact your system administrator for access to using **cron** and **crontab**.

Wait for about 10 minutes, then continue to the next step.

3. Type **crontab -l** and press **Return** to display the contents of your current crontab.
4. Now remove your current crontab by typing **crontab -r** and pressing **Return**. You now have mail in your mailbox from the cron command running your cron entries. The mail messages are viewed later in Module 86.
5. Turn to Module 4 to continue the learning sequence.

Module 27
crypt

DESCRIPTION

The external **crypt** command encrypts and decrypts the standard input and standard output. It reads the standard input and encrypts it based on the supplied encryption key. The encrypted data is written to the standard output. Thus it is usually necessary to redirect the standard output and input to and from files when using **crypt**. To decrypt a file you just reverse the filenames on the command line, allowing **crypt** to read from the encrypted file and writing the decrypted data to a file.

NOTE

System V ships the **crypt** command as part of a security package. Your system may not have the crypt command if this software package was not purchased and installed.

The security of the encrypted data depends on the accessibility to your encryption key and plain data. It is advisable not to use an encryption key on the command line. If an encryption key is placed on the command line, it can be easily viewed by another user browsing the output of the **ps** command. Data that has not been encrypted is obviously at the disposal of whoever can read it.

If you plan on using encrypted files, you should learn to use the encrypt options on the various editors supplied in the UNIX System. This will prevent you from leaving decrypted files in your directory.

COMMAND FORMAT Following is the general format of the **crypt** command.

```
crypt [ key ]
crypt [ -k ]
```

BSD (Berkeley)

```
crypt [ key ]
```

Since **crypt** does not read from files, you must use the shell redirection symbols to control the input and output of data. For instance:

```
crypt key < not_crypted > crypted
crypt key < crypted > not_crypted
```

Options The following option is used to control how **crypt** functions.

-k The -k option causes **crypt** to use the encryption key assigned to the environment variable CRYPTKEY.

BSD (Berkeley)	
-k	Not supported

Arguments The following argument may be passed to the **crypt** command.

key — The key word that controls the transformation of the data. It is used by **crypt** to perform the encryption and decryption.

DIAGNOSTICS AND BUGS If the proper key is not used, **crypt** will decrypt the file and generate various control sequences, possibly causing your terminal or printer to enter strange modes.

RELATED COMMANDS Refer to the **ed**, **ex**, and **vi** commands described in modules 39, 43, and 151.

RELATED FILES The **crypt** command reads from the standard input and writes to the standard output.

APPLICATIONS

You may find it necessary to secure private information on your system. Maybe you're a manager and you need to keep all of your memos confidential. By setting the permissions to read-only by others you can prevent most casual users from viewing the contents. But the super-user can always look at any file, no matter how the permissions are set. To secure the data, you can encrypt it with the **crypt** command. No matter who attempts to read the file, they must have the encryption key to extract the data. Hopefully, only you know the encryption key. It will take someone with more than normal curiosity to break the encrypted data code and read the contents of the file.

WARNING

Although breaking the code to decipher encrypted data is very difficult, it can be done. If someone with encryption experience and enough time trys to break the code to your data, he/she will probably succeed. But the easiest way for your data to be deciphered is by someone acquiring your key. It is advisable not to use your key on the command line or in the CRYPTKEY variable. An experienced systems programmer can acquire these values from your executing shell environment. The value can also be acquired when you type it into the prompt from **crypt**. Also be aware of which crypt program you are executing; make sure it is /bin/crypt, not from your directory or some local bin directory. If it is not /bin/crypt, it may be a trojan horse program which writes your key into another user's log file, allowing the user to access your encrypted data.

TYPICAL OPERATION

In this activity you use the **crypt** command to encrypt a file and then decrypt it back to normal. Begin at the shell prompt.

1. Type **crypt < file1 > file1.e** and press **Return**. Notice you are prompted for a key.

   ```
   cj> crypt < file1 > file1.e
   Enter key:
   ```

   Enter an encryption key. At least six characters are suggested.

2. Type **file file1.e** to see what type of data is stored in the encrypted file. The file type should be *data*.
3. Type **od -c file1.e** and press **Return** to display the encrypted data. Notice the characters look like jibberish.
4. To decrypt the file type **crypt < file1.e > file1**. Enter the encryption key as before when crypt prompts for it.
5. Now let's see if the file is readable by typing **cat file1** and pressing **Return**.
6. Type **rm file1.e** and press **Return**.
7. Turn to Module 98 (SV), Module 122 (BSD) to continue the learning sequence.

Module 28
csplit (SV)

DESCRIPTION

The external **csplit** command divides an input into smaller files based on context or line numbers. The context splitting is performed based on words and patterns you specify on the command line. The line numbers must also be specified as context strings. The input file remains unaltered.

COMMAND FORMAT Following is the general format of the **csplit** command.

```
csplit [ -ks ] -fprefix ] - arg1 [arg2 ... ]
csplit [ -ks ] -fprefix ] file arg1 [ arg2 ... ]
```

Options The following list describes the options and their arguments that may be used to control how **csplit** functions.

-k — All output files created are not removed if an error occurs. The standard procedure for **csplit** is to remove all files it has created if an error occurs.

-s — Suppresses the displaying of character counts. Normally **csplit** displays the character counts for each file it creates.

-f*prefix* — Replaces the *xx* prefix used for output filenames. Must be 12 characters or less in length. Some System V systems implement BSD file systems which allow for longer filenames.

Arguments The following list describes the arguments that may be passed to the **csplit** command.

- — Forces **csplit** to read from the standard input. Useful if you need **csplit** to read the output of a pipe.

file — The input file that is read by **csplit** and divided into smaller files.

arg1 — The strings that are used by **csplit** to divide the input into smaller, more manageable output files. See the discussion in the following Context section.

Context Using **csplit** allows multiple ways to divide input into smaller files. The arguments provided on the command line may be any of the following:

NOTE

Enclose all regular expressions in quotes to prevent interpretation by the shell.

/expr/ — Creates a file containing lines from the current input line up to but not including the location of the regular expression *expr*. You may place a *+n* or *-n* after this argument to specify the number of lines before or after the expression to split the input. No spaces are allowed between the expression and this offset number.

TYPICAL OPERATION

In this activity you use the **crypt** command to encrypt a file and then decrypt it back to normal. Begin at the shell prompt.

1. Type **crypt < file1 > file1.e** and press **Return**. Notice you are prompted for a key.

   ```
   cj> crypt < file1 > file1.e
   Enter key:
   ```

   Enter an encryption key. At least six characters are suggested.

2. Type **file file1.e** to see what type of data is stored in the encrypted file. The file type should be *data*.
3. Type **od -c file1.e** and press **Return** to display the encrypted data. Notice the characters look like jibberish.
4. To decrypt the file type **crypt < file1.e > file1**. Enter the encryption key as before when crypt prompts for it.
5. Now let's see if the file is readable by typing **cat file1** and pressing **Return**.
6. Type **rm file1.e** and press **Return**.
7. Turn to Module 98 (SV), Module 122 (BSD) to continue the learning sequence.

Module 28
csplit (SV)

DESCRIPTION

The external **csplit** command divides an input into smaller files based on context or line numbers. The context splitting is performed based on words and patterns you specify on the command line. The line numbers must also be specified as context strings. The input file remains unaltered.

COMMAND FORMAT Following is the general format of the **csplit** command.

```
csplit [ -ks ] -fprefix ] - arg1 [arg2 ... ]
csplit [ -ks ] -fprefix ] file arg1 [ arg2 ... ]
```

Options The following list describes the options and their arguments that may be used to control how **csplit** functions.

-k — All output files created are not removed if an error occurs. The standard procedure for **csplit** is to remove all files it has created if an error occurs.

-s — Suppresses the displaying of character counts. Normally **csplit** displays the character counts for each file it creates.

-fprefix — Replaces the *xx* prefix used for output filenames. Must be 12 characters or less in length. Some System V systems implement BSD file systems which allow for longer filenames.

Arguments The following list describes the arguments that may be passed to the **csplit** command.

- — Forces **csplit** to read from the standard input. Useful if you need **csplit** to read the output of a pipe.

file — The input file that is read by **csplit** and divided into smaller files.

arg1 — The strings that are used by **csplit** to divide the input into smaller, more manageable output files. See the discussion in the following Context section.

Context Using **csplit** allows multiple ways to divide input into smaller files. The arguments provided on the command line may be any of the following:

NOTE

Enclose all regular expressions in quotes to prevent interpretation by the shell.

/expr/ — Creates a file containing lines from the current input line up to but not including the location of the regular expression *expr*. You may place a *+n* or *-n* after this argument to specify the number of lines before or after the expression to split the input. No spaces are allowed between the expression and this offset number.

The current line is changed to be the new position defined by the expression. The regular expressions recognized by **ed** are used by **csplit**.

%expr% Do not create a file containing lines specified by the regular expression. Same as the */expr/* context but the output is NOT created.

lineno Creates a file from the current position up to but not including *lineno*. *Lineno* must be an integer number.

{num} Repeat preceding argument. You use this argument in conjunction with any of the above arguments. If you use *{num}* after an *expr*, **csplit** uses the argument *num* times. If {num} follows a *lineno*, the input is split every *lineno* lines for *num times*. For example, csplit file 100 {10} would split the file every 100 lines.

FURTHER DISCUSSION The **csplit** command is an enhanced **split** command. It lets you divide a file based on certain criteria. You are not restricted to an equal number of lines per output file as with the **split** command. By specifying regular expression strings as arguments you can divide the input into files based on meaningful sections. The splitting of the input can also be done based on line numbers. Thus **csplit** can perform equal file splitting and context file splitting.

The output is written to files with a prefix and an extension. You can specify the prefix using the -f*prefix* option. If you do not specify a prefix, **csplit** uses the xx prefix as a default. The suffix is also placed on the output files by the **csplit** command. The suffixes begin at 00 and continue to 99. The input is divided into these files based on the following concepts:

xx00 First file contains all lines from the beginning of the file up to but not including the line referenced by the first argument.

xx01 Contains all lines from arg1 (argument one) up to but not including arg2.

xxNN Contains all lines from argNN-1 to the end of the file, where NN is the number of arguments you specified on the command line.

One useful example of the **csplit** command is to divide a large **troff** or **nroff** document into smaller sections. One way to do this is to divide each level 1 heading into a file of its own. For instance, you might try the following command:

```
cj> csplit -k document '/^.H 1/' {99}
```

This will split a file containing troff (nroff) level 1 headings into files containing an entire level 1 heading. The -k option keeps **csplit** from removing all the files it created if there are not 99 occurrences of the ^.H 1 string in the input file.

DIAGNOSTICS AND BUGS One of the messages returned by **csplit** is not very clear, it is:

arg - out of range This message is informing you that one of the arguments you supplied did not locate a position within the allowed range. The allowed range is from the current line to the end of the file.

RELATED COMMANDS Refer to the **split** command described in Module 124. Also see Module 39 on **ed**.

RELATED FILES The **csplit** command writes its output to files with prefixes of xx and suffixes of 00, 01, and so on up to 99.

APPLICATIONS

Using **csplit**, you can divide a file into smaller files using various line numbers and/or strings contained within the input file. This provides you with tremendous flexibility for easily breaking a large file into smaller, more controllable files.

TYPICAL OPERATION

In this activity you use the **csplit** command to split the names of users into three groups based on their login names. Begin at the shell prompt.

1. Type **cut -f1 -d: /etc/passwd | sort | csplit - /^k/ /^t/** and press **Return**. This will divide your system's passwd file into three smaller files. The filenames will be xx00, xx01, and xx02. The 01 file contains login names beginning with a through j, 01 has login names beginning with k through s, and 02 has t through z. If your **csplit** does not support the - argument, an error like "cannot open. . ." is displayed, **cut** and **sort** the usernames to a file, then execute **csplit** with the new filename as its file argument.
2. If you wish to view any of the xxNN files created by **csplit**, type **cat xx00** and press **Return**. Replace 00 with 01 or 02 to display the other files.
3. Remove all files beginning with *xx* by typing **rm xx*** and pressing **Return**.
4. Turn to Module 69 to continue the learning sequence.

ITC "Technicum" after course

43 page

```
int main(int argc, char *argv[])
{
    return;
}
```

Module 29
cut (SV)

DESCRIPTION

The external **cut** command displays selected columns or fields from each line of a file. A column is one character position. A field is a delimiter (tab) separated group of characters. The following functions and features are provided by **cut**:

- Displays columns of a table or file
- Displays fields of a table or file
- A field delimiter may be specified
- Suppress lines with no line delimiters

BSD (Berkeley)

Although **cut** is not usually implemented on BSD systems, you can accomplish the same results using **nawk**.

COMMAND FORMAT Following is the general format of the **cut** command.

```
cut -clist [ file_list ]
cut -flist [ -dchar ] [ -s ] [ file_list ]
```

Options The following list describes the options and their arguments that may be used to control how **cut** functions.

-c*list* — Display (cut) columns, specified in *list*, from the input data. No spaces are allowed within the list. Multiple values must be comma (,) separated. The *list* defines the exact columns to display. For example, the -c1,4,7 notation displays columns 1, 4, and 7 of the input. The -c-10,50- format would display columns 1 through 10 and 50 through end-of-line.

-f*list* — Display (cut) fields, specified in *list*, from the input data. No spaces are allowed within the list. Multiple values must be comma (,) separated. The *list* defines the exact field to display. For example, -f1,4,7 would display fields 1, 4, and 7. The -f2,4-6,8 would display fields 2, 4, 5, 6, and 8.

-d*char* — The character *char* is used as the field delimiter. The default delimiter is a tab character. To use a character that has special meaning to the shell, you must quote the character so the shell does not interpret it. For example, to use a single space as a delimiter, type -d' '.

-s — Suppress lines that do not contain a field delimiter. Normally these lines are displayed.

Arguments The following argument may be passed to the **cut** command.

file_list	One or more files to be cut and displayed.

DIAGNOSTICS AND BUGS The following messages may be returned from **cut** if a problem is encountered.

line too long	A line in the file is longer than 1023 characters or fields. May also be caused from no new-line characters in the file.
bad list for c/f option	Either the *list* for the c or f option is incorrect or no c or f option was specified.
no fields	No *list* was specified for the c or f option.
no delimiter	No *char* was specified for the d option.
cannot handle multiple adjacent backspaces	Multiple backspaces located beside each other cannot be handled correctly by **cut**.
cannot open *file*	Either you do not have read permission to the specified file or the file does not exist.

RELATED COMMANDS Refer to the **grep** command described in Module 60 on selecting rows from a file. Another database type command is **join** discussed in Module 69. Other related commands include the **paste** and **pr** commands discussed in modules 101 and 105.

APPLICATIONS

You can use the **cut** command to project data from a file. That is you can select certain fields or columns and display them. By using shell indirection you can create new files with a subset of columns or fields contained in the first file.

Sometimes **cut** is used in shell programming to select certain values from a line. This is one of many ways to perform such a selection. If you are selecting fields for a value, you should probably use the **set** command where possible and echo the desired positional parameter. Whereas, you may have to use the **cut** command to select values from exact columns.

TYPICAL OPERATION

In this activity you use the **cut** command to display fields out of the /etc/passwd file and cut columns from the output of the **who** command. Begin at the shell prompt.

1. Cut the first field of the *db/phone* file into the *first* file by typing **cut -f1 -d: db/phone > first**. The -d: specifies that the fields are colon (:) separated.
2. If you are using the ksh, press **Escape**, type **kf1r4ffcwcity**, and press **Return**. If you do not have the ksh, type **cut -f4 -d: db/phone > city** and press **Return**.
3. Type **cat first city** and press **Return** to display the two new files.
4. Type **who -u | cut -c-8,43-** to display the usernames of all users currently logged on the system and the PID for their login shell. This assumes your who command displays the standard who information.
5. Turn to Module 101 to continue the learning sequence.

Module 30
date

DESCRIPTION

The **date** command displays the date and time. The super-user (system administrator) can use the date command to change the system's date and time. Any user can use the date command to display the date and time in a standard default format or in a formatted output.

COMMAND FORMAT Following is the general format of the *date* command.

```
date [ -a[-]sss.fff ] [ -u ] [mmdd]hhmm
date [ -a[-]sss.fff ] [ -u ] [ mmddhhmmcc[yy] ]
date [ -u ] [ +format ]
```

BSD (Berkeley)

```
date [ -n ] [ -u ] [ yymmddhhmm[.ss] ]
```

Some BSD systems support the format function of the **date** command. Check your system's User Reference Manual.

Options The following list describes the options that may be used to control how **date** functions.

-a[-]*sss.fff*	Adjust the time by plus or minus ([-]) *sss* seconds and *fff* fractions of a second. The systems clock will speed up or slow down until it has slowly adjusted by the specified time of *sss.fff*. This option is not supported on all systems.
-u	Display the date in GMT (Universal time). The system's actual time is set in GMT. The **date** command performs the necessary conversion to display the date in your current time zone, based on the TZ environment variable. The -u option can also be used to set the time in GMT.
mmddhhmm[cc]yy	Change the date and time to the new date and time specified. The century (cc) is not required. The first *mm* specifies the month, *dd* the day, *hh* the hour (24-hour base), the second *mm* the minutes, and *yy* specifies the year.
+format	Special format strings used to retrieve the date and time in formatted output. See the following format descriptors.

BSD (Berkeley)

yymmddhhmm[.ss]	Change the date and time to the new date and time specified. The *yy* specifies the year, the first *mm* the month, *dd* the day, *hh* the hour (24-hour base), and the second *mm* the minutes. If you include the *.ss* then the seconds are set appropriately.

-n	If *timed*, the network time daemon, is running, **date** sets the time globally on all systems on the local network unless you specify the -n option. The *timed* command synchronizes the clocks of machines in a local area network.

Format Descriptors The following is a list of format descriptors. Each descriptor must be preceded by a % character. To display a % character enter %%.

Date format descriptors:

%A	The full weekday (Sunday-Saturday)
%a	The abbreviated weekday (Sun - Sat)
%B	The full month name (January-December)
%b	The same as %h
%c	The country specific date and time format
%D	The date (mm/dd/yy)
%d	The day of the month (01 - 31)
%e	The day of the month (1-31; single digits are preceded by blanks)
%h	The abbreviated month (Jan - Dec)
%j	The julian date, day of the year (001 to 365)
%m	The month of the year (01 - 12)
%U	The week number of year (00-53; Sunday is first day of week)
%W	The week number of year (00-53; Monday is first day of week)
%w	The day of the week (0=Sunday - 6=Saturday)
%X	The country specific date format
%Y	The year in century format(1990)
%y	The current year (00 - 99)

Time format descriptors:

%H	The hours (00 - 24)
%I	The hours (00 - 12)
%M	The minutes (00 - 59)
%p	The string containing ante or post meridian indicator (defaults are AM or PM)
%r	The time (HH:MM:SS AM/PM)
%S	The seconds (00 - 59)
%T	The time (HH:MM:SS)
%x	The country specific time format
%Z	Timezone name

Output format characters:

%n	Display a new-line
%t	Display a tab
%%	Display a percent sign (%)

BSD (Berkeley)

Most BSD systems do not support any format descriptors.

DIAGNOSTICS & BUGS The following list describes possible error messages you may receive from **date**.

no permissions	You tried to change the date and you are not the super-user.
bad conversion	Syntax error in new date or time specification
bad format character	Syntax error in the format string

RELATED FILES The **date** command writes to the standard output.

APPLICATION

The **date** utility can be used to change the date or display the date and time. It can produce variable formatted output. If the date is needed in an **echo** command, you can use the date command to perform most echo command functions. It is often used in shell programming to provide checks, filenames, and information pertaining to the date or time.

TYPICAL OPERATION

In this activity you use date to display the current date and time and then display a formatted date and time.

1. Type **date** and press **Return**. Notice the standard format that date uses to return the information.

```
cj> date
Tue Dec 13 20:23:40 CST 1988
```

System V

2. Type the following and press **Return**. Notice that date can "echo" other information.

```
cj> date +"It is %T%ton%t%D"
It is 20:27:12      on      12/13/88
```

NOTE

It is advisable to enclose all of the date arguments in single quotes to prevent the shell from interpreting the format string. If you use shell variables in the format string then use double quotes in place of single quotes.

---

3. Turn to Module 93 (SV), Module 156 (BSD) to continue the learning sequence.

Module 31
dd

DESCRIPTION

The external **dd** command converts and copies files. It can read input from a file or from the standard input. It writes to a file or to standard output. The **dd** command lets you copy data, from one device to another, that do not have the same block size. For example, you might have a tape that has blocks containing 1024K bytes while your file system has 512-byte blocks. In some cases the **cpio** and **tar** commands might fail. You could use **dd** in front of the appropriate command to extract the data from the tape and convert it to a block size the other commands can understand.

It provides several special services for converting data:

- Reads one block size and writes another block size
- Swaps the high order and low order bytes
- Converts EBCDIC characters to ASCII characters
- Converts ASCII characters to EBCDIC characters
- Converts lowercase to uppercase
- Converts uppercase to lowercase

It also provides several services for copying data.

- Skips blocks on input
- Skips blocks on output
- Copies a specified number of blocks
- Pads input blocks

COMMAND FORMAT Following is the general format of the **dd** command.

```
dd [ option=value ... ]
```

Options The following list describes the options and their arguments that may be used to control how **dd** functions.

Option	*Description of Value*
bs=*n*	Sets the input and output block size to *n* bytes. The bs option overrides the ibs and obs options.
cbs=*n*	Sets the conversion buffer size to *n* bytes. Applies only to the ASCII/EBCDIC/IBM and (BLOCK/UNBLOCK in BSD) conversions. If you are converting from EBCDIC to ASCII, *n* bytes of an input block are read into the cbs buffer. Trailing spaces are removed and a new-line is added. The contents of the cbs buffer are then written to the output. If you are converting from ASCII to EBCDIC, an ASCII line is read into the cbs buffer. The new-line is removed and trailing blanks are added to create a block containing *n* bytes. For example, if you want to convert IBM records containing 132 columns, you specify cbs=132.

This causes input blocks to be subdivided into 132-byte sections. These sections are equated to ASCII lines.

conv=ascii — Converts EBCDIC input to ASCII output.

conv=block — Converts variable length records to fixed length records. You must set obs to desired output block length. This pads records of different lengths with spaces to all be the same length.

conv=unblock — Converts fixed length records to variable length records. The ibs must be set but the output records are stripped of all trailing blanks, thus creating variable length records.

conv=ebcdic — Converts ASCII input to EBCDIC output.

conv=ibm — Converts ASCII to a less common EBCDIC format.

conv=lcase — Converts uppercase characters to lowercase characters.

conv=ucase — Converts lowercase characters to uppercase characters.

conv=swab — Swaps the odd and even byte in each pair of bytes. Useful to switch bytes between systems where one uses high order first and another uses the low order byte first.

conv=noerror — If an error occurs **dd** continues to process. Normally, if there is a problem, **dd** aborts immediately following the error.

conv=sync — Pads each input record with spaces until the ibs buffer is filled.

conv=*val1*, *val2*,... — Multiple values may be combined on the conv option statement by separating the values with commas. For example,

```
dd conv=lcase,ascii
```

converts the standard input from uppercase to lowercase and converts the characters from EBCDIC to ASCII.

count=*n* — Allows only *n* blocks of input to be copied.

ibs=*n* — The input block size is set to *n* bytes. If the ibs option is not specified, a 512-byte block is used.

if=*file* — Specifies *file* as the input file. If the if option is not specified, the standard input is used.

obs=*n* — The output block size is set to *n* bytes. If the obs option is not given, a 512-byte block is used.

of=*file* — Specifies *file* as the output file. If the of option is not used, the standard output is used.

seek=*n* — Seek *n* blocks forward from the beginning of the output file. This means **dd** skips over the first *n* blocks of the output file before it starts to write the data.

iseek=*n* — Seek *n* blocks forward from the beginning of the input before copying data.

oseek=*n* — Same as seek.

skip=*n* — Skips forward the first *n* blocks of the input file.

BSD (Berkeley)	
iseek	Not supported
oseek	Not supported

In the previous options the *n* count can have a suffix attached to indicate the block size. The suffix may be one of the units listed in the following notations.

*n*b	Specifies the block size in increments of 512 bytes. For example, 4b means 4 * 512 which equals 2048 bytes, so the block size is 2048 bytes.
*n*k	Specifies the block size in increments of 1024 bytes.
*n*w	Specifies the block size in increments of 2 bytes (words).

Arguments The **dd** command has no arguments, although you can use the shell's indirection feature to provide input from a file and output to a file. For example,

```
dd conv=ascii < EBCDICfile > ASCIIfile
```

converts the contents of the file named *EBCDICfile* from EBCDIC format to ASCII format and stores the output in the file named *ASCIIfile*.

DIAGNOSTICS AND BUGS After it completes the requested process, the **dd** command returns the number of full and partial blocks read in and written out.

The BLOCK/UNBLOCK conversions cannot be combined with the ASCII/EBCDIC/IBM conversion options.

RELATED COMMANDS Refer to the **tr** command described in Module 140.

RELATED FILES The **dd** command reads from the standard input and writes to the standard output unless you specify files using options.

APPLICATIONS

The most common use of **dd** is to copy the entire contents of one device to another device. For example, if you wanted to move a file system from one disk device to another, you could use the **dd** command. Another use is inputting EBCDIC data in from an IBM Mainframe tape to the UNIX system. For example,

```
dd if=/dev/mt/rmt0 of=emp.list ibs=1320 cbs=132 conv=lcase,ascii
```

copies the input from an IBM tape mounted on the /dev/mt/rmt0 device tape drive. The output is placed in the emp.list file. The input is in 132-character records with 10 records per block. **dd** also converts the input to lowercase and to ASCII.

TYPICAL OPERATION

In this activity you use the **dd** command to convert an ASCII file to EBCDIC, display the EBCDIC file with the **cat** command and then convert it back to ASCII, and display the output to your screen. Begin at the shell prompt.

1. Type **dd if=file1 of=EBCDIC conv=ebcdic** and press **Return**. This converts the contents of file1 to EBCDIC and stores the EBCDIC data in the file named EBCDIC.

2. Type **cat -v EBCDIC** and press **Return**. Notice garbage output is sent to your screen. This is the EBCDIC characters (bytes) being interpreted by **cat** as ASCII characters, which they aren't and thus do not display correctly.
3. Now type **dd if=EBCDIC conv=ascii,ucase** and press **Return**. This converts the contents back to ASCII and to uppercase characters.
4. Type **rm EBCDIC** and press **Return** to remove the ebcdic data file.
5. Turn to Module 126 to continue the learning sequence.

Module 32
df

DESCRIPTION

The external **df** command displays the disk free space. It is used to find out how much disk space is available in each mounted file system. The **df** command is used to:

- Display available disk blocks and inodes(files).
- Display total disk blocks and inodes.
- Display available disk blocks without inode count.
- Display the device on which the filesystem is mounted.
- Display total kilobytes in a filesystem.
- Display used kilobytes in a filesystem.
- Display available (free) kilobytes in a filesystem.
- Display percent used of kilobytes and inodes.

BSD (Berkeley)

The BSD version of df supports the same features as System V except for block counts.

COMMAND FORMAT Following is the general format of the **df** command.

```
df [ -F FStype ] [ -begklntV ] [ SVoptions ] [ -o FSoptions ] filesystem
df [ -F FStype ] [ -begklntV ] [ SVoptions ] [ -o FSoptions ] mount_point
df [ -F FStype ] [ -begklntV ] [ SVoptions ] [ -o FSoptions ] resource
```

BSD (Berkeley)

```
df [ -ai ] [ -t type ] [ filesystem ]
df [ -ai ] [ -t type ] [ file ]
```

Options The following options may be used to control how **df** functions.

-F *FStype* — Specifies the type of file system. Only used for unmounted file systems. The *FStype* should be specified with the -F option or be found in the /etc/vfstab by matching *mount_point*, *special*, or *resource*. Supported types are s5, ufs, nfs, efs, ffs, and rfs. Not all of these types may be supported on all systems.

-b — Display only the number of free kilobytes.

-e — Display only the number of free files.

-g	Display the entire statvfs structure. Used only for mounted file systems. Cannot use with -beknt options, -o or the SV options.
-k	Display allocation information in kilobytes. Should be used by itself since it uses a different format.
-l	Display information for local file systems only. Used only for mounted file systems. Cannot use with -o or SVoptions.
-n	Display on the FStype name. If no arguments, a list of all mounted file system types is displayed. Used only for mounted file systems. Cannot use with -o or SVoptions.
-t	Displays a total listing. Displays the total number of blocks and inodes and the free or available blocks and inodes. Overrides the -ben options.
-V	Verify the command line. The command line is echoed back to the screen after **df** has added information from the /etc/mnttab and /etc/vfstab.
-o *FSoptions*	Used to specify FStype specific options. The FSoptions are a comma-separated list of file system specific options. See your system's User's Reference Manual **df** command man pages.
-i	Displays the number of free and used inodes.
SVoptions	These are System V specific options. Refer to your systems df (s5) man page.

BSD (Berkeley)

-i	Displays the number of free and used inodes.
-t *type*	Displays only filesystems of a given type. The types are nfs and 4.2. Type nfs is for a Network File System and 4.2 is for a local BSD File System.

Arguments The following list describes the arguments that may be passed to the **df** command.

filesystem	The filesystem device name. The name of the device where the filesystem resides physically. Such as /dev/da0a or /dev/dsk/c0d0s0.
mount_point	The name of the directory where the filesystem is mounted. Referred to as the filesystem mount point.
resource	An NFS or RFS resource name.

If no arguments are given then all filesystems currently mounted are reported.

BSD (Berkeley)

file	The name of a file within a filesystem. The disk space for the filesystem containing *file* is displayed. May be a directory, such as the filesystem mount_point.

FURTHER DISCUSSION A file system is a set of files and directories stored on one physical part of a disk. A device name (/dev/dsk/0s0) or a logical directory name may be used by **df** to access a file system. For a file system to be checked via its logical name it must be mounted. Mounting a file system makes it available for use. The system becomes aware of the physical media and relates a logical name to it.

In general, the rule of thumb for available space on a file system is 10 percent of the maximum size. If you have a file system that is more than 90 percent full, you should remove all files no longer needed. The system administrator is responsible for keeping file systems within the set guidelines.

BSD (Berkeley)

The BSD system reserves a portion of each file system for allocation routines to work properly. The reserved part is usually 10 percent but may be adjusted by using the super-user command **tunefs**.

Once a file system reaches 100 percent full, only the super-user can allocate more space.

Alleviating Full Disks To alleviate the problem, the administrator may use the **find** and **cpio** commands to move an entire directory structure to a new file system. Another alternative is to have users mv files to a directory in their home directory named ARC. Then run a job that saves all user files stored in ARC directories to tape. Of course, a satisfactory solution is to just have the users remove all unnecessary files.

RELATED COMMANDS Refer to the **du** command described in Module 37.

RELATED FILES The following files are read and used by the **df** command.

/etc/checklist	A list of all normally mounted filesystems (SV only)
/dev/dsk/*	The disk partitions where filesystems reside (SV only)
/dev/d*	The disk partitions where filesystems reside (BSD only)
/etc/mnttab	A list of all currently mounted filesystems (SV only)
/etc/mtab	A list of all currently mounted filesystems (BSD only)
/etc/fstab	A list of all normally mounted filesystems (BSD only)
/etc/vfstab	A list of default parameters for each file system (SV only)

APPLICATIONS

You can use the **df** command to check available disk space. The numbers returned by the System V **df** command are blocks, not bytes. Therefore, don't forget to multiply by 512, 1024, or 2048, depending on your system's block size.

If you're low on free space, you may want to remove all unnecessary files and send a mail note to the system administrator stating your observance. The amount of available space kept for a buffer is dependent upon each site's requirements. Some system administrators require 10000 blocks while others set a minimum of 500. The size is heavily dependent on the type of work being done on each filesystem and the size of the filesystem.

TYPICAL OPERATION

In this activity you use the **df** command to list the available disk space on you system. Begin at the shell prompt.

System V Only

1. Type **df -t** and press **Return** to display the total space on the system and the total available space. You screen will look something like this:

```
cj> df -t
/       (/dev/dsk/c1d0s0):      1743 blocks   1398 inodes
                   total:     18144 blocks   2256 inodes
/usr    (/dev/dsk/c1d0s2):     11364 blocks   8752 inodes
                   total:    121176 blocks  15136 inodes
/u1     (/dev/dsk/c1d0s5):     33023 blocks   2103 inodes
                   total:    181934 blocks  18321 inodes
```

The 1743 refers to free blocks; the 1398 is free inodes. The 18144 refers to total blocks and the 2256 is total inodes. The output of some vendors' **df** command varies from the standard.

BSD (Berkeley) Only

1. Type **df** and press **Return**. The output resembles the following.

```
cj> df
Filesystem          kbytes    used   avail   capacity  Mounted on
/dev/da0a            20143   17608     520      97%    /
/dev/da0e           122559   76660   33643      69%    /usr
/dev/da1g           183887  105277   60221      64%    /u1
```

---

2. Turn to Module 37 to continue the learning sequence.

Module 33
diff

DESCRIPTION

The external **diff** command compares two text files for differences. It determines which lines must be changed to make the two files identical. The **diff** command scans the two files and indicates editing changes that must be made to the first file to make it identical to the second file. The changes can be saved for use as an **ed** command script to change the first file. It can also compare directories. System V provides a second command to compare directories; refer to the **dircmp** command.

COMMAND FORMAT Following is the general format of the **diff** command.

```
diff [ -lrs ] [ -S name ] [ -cefhn ] [ -biwt ] dir1 dir2

diff [ -cefhn ] [ -biwt ] file1 file2
diff [ -cefhn ] [ -biwt ] - file2
diff [ -cefhn ] [ -biwt ] file1 -

diff [ -D string ] [ -biw ] file1 file2
diff [ -D string ] [ -biw ] - file2
diff [ -D string ] [ -biw ] file1 -

diff [ -C string ] [ -biw ] file1 file2
diff [ -C string ] [ -biw ] - file2
diff [ -C string ] [ -biw ] file1 -
```

You may want to think of file1 as the old file and file2 as the new file.

Options The following list describes the options and their arguments that may be used to control how **diff** functions.

Comparison control options:

-b Causes blanks (spaces and tabs) to compare equally even if an unequal number of blanks exist. All trailing blanks are ignored. For example, if the first line was in file1 and the second line was in file2, they would compare as equals.

```
file1:  A      sample line of text      here
file2:  A sample  line of text here
```

-i Causes the case of letters to be ignored. For example,

THE BIG dog ran fast.

and

The big dog ran fast.

match as equal lines.

-t Expands tabs on input to spaces on output. Normal output adds additional characters to the front of each line. This may change the indention of the original text, making it difficult to read. This option preserves the original text indention. The -c option also adds additional characters, causing indention problems.

-w Causes all white spaces (blanks and tabs) to be ignored. For example,

```
if ( x == y )
```

and

```
if(x==y)
```

compare as equals.

Directory comparison options:

-l Display long output listing. The ouput is piped through **pr** for pagination. Other differences are saved and summarized after all text file differences are displayed.

-r Recursively descends through subdirectories.

-s Display files that are the same. Normally, identical files are not displayed.

-S *name* Begin the directory comparison with file *name*. Normally, all files in the directory are compared.

Mutually exclusive options:

-D*string* Creates a merged version of *file1* and *file2* on the standard ouput. C preprocessor controls are included in the output. If the *ifdef string* is not defined then a compile (cc) on the output would yield the same program as a compile on *file1*. If the *ifdef string* is defined then a compile would be the same as a compile on *file2*.

-c[*n*] Displays a comparison with *n* lines of context. The default for *n* is 3. The output begins with the identification and creation dates of each file. Each change encountered is separated with a dozen asterisks (*). Lines removed from *file1* are preceded by a hyphen (-). Lines removed from *file2* are preceded by a plus (+). Lines changed from *file1* to *file2* and vice versa are preceded with an exclamation mark (!). System V does not support the [n]. See the -C[n] option for variable context sizes.

-C[n] System V only. Same as the -c[*n*] option on BSD.

-e Produces an **ed** script consisting of the *a* (append), *c* (change), and *d* (delete) commands. These commands can be used as input to **ed** to change *file1* to match *file2*. See the following section on Version Control.

-f Produces a script similar to that produced by -e, but the order is for *file1* from *file2*. These commands are not usable with **ed**.

-h Does not attempt to find the most efficient way to edit the changes. It is fast, but not thorough. The changes must be short and well separated. It does work on files of unlimited size. The -e and -f options are disabled if -h is specified.

-n Produces a script similar to the -e option. The order is reversed. Each insert or delete command contains a count of changed lines. This format is used by **rcsdiff**.

Arguments The following list describes the arguments that may be passed to the **diff** command.

file1 The first input file used in the comparison. If *file1* is a directory name, the *file2* file in directory *file1* is used for comparison. For example, if you specify,

```
diff adir afile
```

diff uses *adir/afile afile* for the two files.

file2 The second input file used in the comparison. If *file2* is a directory, the second file is set to *file2/file1*.

- A hyphen may be used in place of either *file1* or *file2* to represent the standard input. This allows you to pipe input to **diff** redirect input from a file, or type input from your keyboard for comparison.

dir1 The first directory containing files used for comparison.

dir2 The second directory containing files used for comparison.

DIRECTORY COMPARISON If both arguments are directories, **diff** sorts the contents of each directory, then performs normal **diff** comparisons on files. Binary files that differ, common subdirectories, and files residing in only one directory are displayed.

OUTPUT FORMAT The output of **diff** varies depending on the options you specify. The *n* represents line numbers and *line* represents the text on a line. The greater than and less than symbols are used to distinguish which lines belong in which file. For example,

```
< Always a line from file1.
---
> Always a line from file2.
```

shows that the first line resides in file1, the --- separates the two files' output, and the third line resides in file2.

Appending The append string informs you which strings exist in *file2* but not in *file1*. Therefore, the lines in *file2* must be appended after the displayed line in *file1*.

```
n1an2,n3    <line n2
...
<line n3
```

The first line informs you to append lines *n2* through *n3* from *file2* after line *n1* in *file1*. The lines of text following the < are to be copied from *file2* and placed in *file1* after line *n1*.

Changing The change string informs you which strings in *file1* must be changed to make the two files agree. The change is basically a delete and append command. It replaces entire lines in *file1* with entire lines from *file2*.

n1,n2cn3,n4 Informs you to replace lines *n1* through *n2* in *file1* with
<line n1 lines *n3* through *n4* from *file2*.
...

<line n2
—
>line n3
...
>line n4

Deleting The delete string informs you which strings do not exist in *file2* but do exist in *file1*. The lines in *file1* must be deleted to make the files conform.

*n1,n2*d*n3*	Informs you to delete lines *n1* through *n2* in *file1*. *File1*
<line n1	is identical to *file2* up to line *n3* in *file2*.
...	
<line n2	

VERSION CONTROL The following steps create an original file and version control files for each updated version of the original. You will only need to keep the original file and a version file for each version.

You must first create your original file.

```
cj> vi original
```

When you want to update the original you simply make a copy and edit the new file (named *new*). Make your changes to the *new* file and save it. Then execute **diff** to make a version control file. The commands are:

```
cj> cp original new
    vi new
    diff -e original new > v01
    rm new
```

Now you have the original and the control file for version 1.

To create a new version as you need to you can write a script named **vs**. The commands for **vs** are:

```
cj> cat vs
(shift ; cat $1 ; echo '1,$p' ) | ed -s$1 > new
vi new
diff -e $1 new > $3
rm -f new
```

On BSD and SV prior to R4 use a – instead of –s on the **ed** command.

To create a new version (version 2) simply use your **vs** script,

```
cj> vs original v01 v02
```

To make a copy of an existing version you simply execute the following **diff** command.

```
cj> diff -e original vXX > vXX.out
```

DIAGNOSTICS AND BUGS The output scripts produced by the -e and -f options may contain periods (.), which cause the **ed** editor to change to command mode. If **diff** meant for the . to be part of the text, the changed file will not be correct.

RELATED COMMANDS Refer to the **cmp** and **comm** commands described in modules 20 and 22.

RELATED FILES The **diff** command reads input from two files. The hyphen (-) may be used on the command line to represent the standard input, thus allowing input to be from a pipe, indirection, or your keyboard.

RETURN CODES The following returned codes are returned by **diff**. You may want to use these codes in shell scripts to verify two files are identical.

0	No differences found.
1	Differences where found.
2	Errors occurred during execution.

APPLICATIONS

The basic use of **diff** is to find text differences between two files. You can specify the -e option to generate usable **ed** scripts. Then you can invoke **ed** to make the necessary changes to the appropriate file.

The **diff** command proves to be useful in comparing versions of files. You may find it helpful in comparing versions of a letter or program source file.

TYPICAL OPERATION

In this activity you use the **diff** command to compare two small files. Begin at the shell prompt.

1. Type **diff file1 file2** and press **Return**. Notice the output strings that inform you how to change file1 to make it match file2.

   ```
   cj> diff file1 file2
   ```

   All the lines of file2 would have to be added to file1 to make file1 identical to file2.

2. Turn to Module 34 (SV), Module 27 (BSD) to continue the learning sequence.

Module 34
dircmp (SV)

DESCRIPTION

The external **dircmp** command compares directories. It searches the two directories you specify for matching filenames. If a filename exists in both directories the two files are compared based on the **diff** command. A list is generated indicating if files with the same name are identical. If a file does not appear in both directories, it is listed as a unique file.

```
cj> dircmp dir1 dir2 | uniq

Nov 12 16:41 1989 dir1 only and dir2 only Page 1

    ./misc

Nov 12 16:41 1989 Comparison of bin misc Page 1

directory.
different./vs
```

The first listing "dir1 only and dir2 only" states which files and directories are only in the dir1 directory and which are only in the dir2 directory. If the names are left-justified, they are in the dir1 directory. If they are preceded by a tab, they are only in the dir2 directory.

The second list "Comparison of dir1 dir2" is a comparison of files with the same name in the bin and misc directories. There are three possibilities in this listing. They are

directory	*name*
different	*name*
same	*name*

The first signifies the two names are directories. The second states the contents of the two files are different. The third states the contents of the two files are the same.

BSD (Berkeley)

Refer to the **diff** command for comparing directories on BSD systems.

COMMAND FORMAT Following is the general format of the **dircmp** command.

```
dircmp [ -ds ] [ -wn ] dir1 dir2
```

Options The following list describes the options and their arguments that may be used to control how **dircmp** functions.

-d If both directories contain a file with the same name, the two files are compared using the **diff** command with no options.

-s Suppress messages about identical files.

-w*n* Set the width of the output lines to be *n* characters. If -w*n* is not specified, the width is set to 72 characters.

Arguments The following list describes the arguments that may be passed to the **dircmp** command.

dir1 The directory to be searched for unique filenames not contained in *dir2* and for filenames that are in *dir2*.

dir2 The directory to be searched for unique filenames not contained in *dir1* and for filenames that are in *dir1*.

RELATED COMMANDS Refer to the **diff** command described in Module 33.

RELATED FILES The **dircmp** command reads the directory files *dir1* and *dir2*, then reads and compares files that are common to both directories. It writes output to the standard output.

APPLICATIONS

The **dircmp** command is used to compare two directories for similarities or differences. Thus you can use it to clean up a duplicate directory. You may want to use it to update a backup directory. It provides an easy way to compare multiple files with only one command.

TYPICAL OPERATION

In this activity you use the **dircmp** command to compare two directories. Begin at the shell prompt.

1. Type **dircmp letters db | uniq** and press **Return**. The **uniq** command simply reduces all duplicate blank lines to one line.

```
cj> dircmp letters db | uniq

Nov 12 16:41 1989 letters only and db only Page 1

                                        ./file2
                                        ./phone

Nov 12 16:41 1989 Comparison of letters db Page 1

directory     .
different     ./file1
```

The first listing "letters only and db only" states which files and directories are only in the *letters* directory and which are in the *db* directory. If the names are left-justified, they are in the *letters* directory. If they are preceded by a tab, they are only in the *db* directory.

The second list "Comparison of letters db" is a comparison of files with the same name in the *letters* and *db* directory.

2. Turn to Module 27 to continue the learning sequence.

Module 35
dirname (SV)

DESCRIPTION

The external **dirname** command parses the given path and returns only the pathname. Its cousin command **basename** returns the filename. Any suffix after the last / (slash) is deleted from the path, including the /. For instance, the command

```
cj> dirname /u1/ts/mylogin/junk.c
```

returns

```
/u1/ts/mylogin
```

The entire path (/u1/tech/mylogin) is returned but the filename junk.c is removed, along with the / separating the parent path and the filename.

COMMAND FORMAT The format of the **dirname** command follows.

```
dirname path
```

Arguments Only one argument may be passed to the **dirname** command.

path The path that is parsed and only the pathname is returned. The filename is removed.

RELATED COMMANDS Refer to the **basename** command described in Module 18.

RELATED FILES The **dirname** command writes to the standard output.

APPLICATIONS

The **dirname** command is normally used as a substitution command within a shell script. It can be used to reduce a full path parameter or variable down to a path without the filename.

TYPICAL OPERATION

In this activity you use the **dirname** command to find the directory of a command. Begin at the shell prompt.

1. Type **DIR=‘dirname /usr/local/bin/newcmd‘** and press **Return**. The DIR variable will be assigned the value of /usr/local/bin. The newcmd file is truncated from the pathname.
2. Type **echo $DIR** and press **Return**. Notice that the path /usr/local/bin is displayed.
3. Turn to Module 91 to continue the learning sequence.

ls -al

Module 36
dirs (csh)

DESCRIPTION

The internal **dirs** command displays the directory stack created by the **pushd** command. The **dirs** command does not manipulate the directory stack. It only displays the contents. You will find it useful if you use the **pushd** and **popd** commands.

COMMAND FORMAT The format of the *dirs* command follows.

```
dirs
```

TIP Most users of the **csh** alias the **pushd**, **popd**, and **dirs** commands. For example, you might want to add the following lines to your .cshrc file.

```
alias u pushd  # use u instead of pushd
alias o popd   # use o instead of popd
alias d dirs   # use d instead of dirs
```

NOTE

Some sites implement the pushd, popd, and dirs command for the ksh. Ask your system administrator, if you use ksh, if the commands are available.

RELATED COMMANDS Refer to the **popd** and **pushd** commands described in modules 104 and 110.

APPLICATIONS

The **dirs** command is simply used to display the directory stack without manipulating it. The directory stack contains a list of directories you can move between by using the **pushd** or **popd** commands.

TYPICAL OPERATION

In this activity you use the **dirs** command to display the directory stack. Begin at the shell prompt.

1. Type **dirs** and press **Return**. Notice the directory stack created with the **pushd** command is displayed.

```
cj> dirs
~ /usr/spool/uucppublic /usr/lib
```

2. Turn to Module 8 to continue the learning sequence.

Module 37
du

DESCRIPTION

The external **du** command is used to find out how much disk space is used in a directory structure. By using the **du** command you can:

- List the disk space used in blocks by each file.
- List the disk space used in blocks for each subdirectory.

The **du** command is somewhat of a counterpart to the **df** command. The **df** informs you that you are low on space. The **du** command tells you which files are using the most disk space.

If you do not provide any options or arguments, **du** displays the number of blocks contained in each file or subdirectory. All files in all subdirectories are added and displayed as one summary for each directory.

If a file is a link, it is only displayed once.

COMMAND FORMAT Following is the general format of the **du** command.

```
du [ -rs ] [ directory_list ]
du [ -ars ] [ file_list directory_list ]
```

Options The following options may be used to control how **du** functions.

-a Allows the use of filenames instead of just directory names.

-r Returns a message for files that cannot be read or opened. Usually, **du** ignores files it cannot read or open. It does not mention them or estimate their size.

-s Summary listing. Displays only the totals for specified directories. If the -a option is used, totals for files are given.

If no options are given, **du** displays the size of each subdirectory in blocks.

BSD (Berkeley)

The BSD version of **du** does not support the -r option.

Arguments The following arguments may be passed to the **du** command.

directory_list One or more directories are searched and the size of each file or subdirectory is displayed.

file_list One or more files are read and the file size displayed.

If no arguments are provided, the current directory (.) is used as an argument.

DIAGNOSTICS AND BUGS The **du** command does not display any information about files that it cannot read or open. If you do not specify the -a option, **du** ignores filenames given in its argument list. If links span across unrelated directories, then the file is counted for each occurrence of a link.

RELATED COMMANDS Refer to the **df** command described in Module 32.

APPLICATIONS

Use the **du** command to check which directories are using most of your disk space. If you are running low on space and need to remove files, the **du** command provides a means of locating where the bulk of your file space is being used.

As a user you can search your own directories for large files and subdirectories. The system administrator can use **du** to locate users that are using large amounts of disk space.

Once you locate the unwanted files, use the **rm** command to discard the files and recover the file space. You may wish to archive the files with the cpio or tar commands before removing them.

TYPICAL OPERATION

In this activity you use the **du** command to display the disk space you are currently using. Begin at the shell prompt.

1. Type **du** and press **Return**. Your screen should resemble the following display:

```
cj du
4       ./letters
2       ./bin
8       ./db
28      .
```

2. Turn to Module 136 to continue the learning sequence.

Module 38
echo

DESCRIPTION

The internal **echo** command displays its arguments to your terminal. More precisely, **echo** writes its arguments on the standard output. The arguments are strings separated by blanks and terminated by a new-line. **echo** understands the more popular C-like escape character sequences. Some of the features and uses of echo are:

- Display strings of text.
- Display variables.
- Display output with various formatting characters (SV only).
- Display ASCII characters based on octal escape sequences (SV only).
- Display menu screens for shell scripts.
- Display prompts for input to shell scripts.

COMMAND FORMAT Following is the general format of the **echo** command.

```
echo [ -n ] [ arg ... ]
```

Options The following list describes the options and their arguments that may be used to control how **echo** functions.

-n Causes echo not to output a new-line character. Thus the text is displayed and the cursor remains after the last displayed character. For example:

```
echo -n "Enter choice: "
```

produces,

```
Enter choice: _
```

whereas

```
echo "Enter choice: "
```

produces,

```
Enter choice:
_
```

Basically you use the -n for shell script prompt lines.

The -n is also supported by the **ksh** print command. So if you have **echo** as an alias for **print** - the -n option will work just like on BSD and csh **echo**.

Arguments The arguments to **echo** are whatever text you wish to write to the standard output. If no arguments are specified, then **echo** displays a blank line by writing a single new-line character. You can use the following list of escape sequence characters to produce special output from **echo**:

CAUTION

You must remember that the shell interprets the \ (backslash) as a special character. Therefore, it is recommended that you place your echo arguments inside double quotes.

\b	produces a backspace on the output.
\c	writes the output, but does NOT terminate it with a new-line. Must be last.
\f	produces a form feed.
\n	produces a new-line.
\r	produces a carriage return.
\t	produces a tab.
\v	produces a vertical tab.
\\	produces a backslash.
\0*n*	produces the ASCII character with the octal code of *n*, where *n* is a 1-, 2-, or 3-digit octal number representing the desired character.

BSD (Berkeley)

The BSD and csh versions of **echo** do not support the escape sequences listed above.

FURTHER DISCUSSION The following examples show some of the useful aspects of this simple command:

```
cj echo *              # display all files in current directory

cj echo *.[ch]         # display all files ending with a .c or .h

cj echo $TERM          # display the value of the TERM variable
```

DIAGNOSTICS AND BUGS When you use the \0*n* escape sequence you must always precede the *n* with a 0 (zero). This informs echo that the number is octal.

To use escape sequences you must either enclose them in double quotes, single quotes, or use two \ (backslashes).

If you need to use these escape characters, the octal equivalents can be found in Appendix G.

RELATED COMMANDS Refer to the **print** command described in Module 106 and the **ksh** command described in Module 71.

RELATED FILES The **echo** command writes to the standard output. Therefore, you may redirect it to a file or a pipe.

APPLICATIONS

The **echo** command is used to display text and variables to the user's terminal. Since the standard output is the default, output may be redirected as desired. It is often used to direct known output into a pipe. It is used in shell scripts to display return codes and debugging output. Menus and data entry screens can be built using **echo**. The **select** command should also be studied for menu building.

TYPICAL OPERATION

In this activity you use the **echo** command to display text and variables. You also use it to output new-lines, tabs, and no new-line via escape sequences. Begin at the shell prompt.

C Shell

If you are using the **csh**, type **sh** and press **Return** to enter a Bourne shell for the following examples. Eliminate the "--" and escape sequences from the **echo** commands if present. Replace the "\t" notation with tabs.

---

1. Type **echo "My home directory is $HOME"** and press **Return**.
2. Type **echo "`tput clear`\n\n\t\t\t$LOGNAME Command Menu"** and press **Return**. Your screen should resemble the following:

```
cj echo "`tput clear`\n\n\t\t\t${LOGNAME} Command Menu"
```

The screen clears and the line is displayed:

```
mylogin Command Menu
```

C Shell

Type **echo "`clear`^V^M^V^M^I^I^I$USER Command Menu"** and press **Return**.

---

3. Type **echo "Press Return to continue: \c"** and press **Return**. Notice the cursor remains at the end of the output string. Your cursor prompt is placed at the end of the string. The \c informs **echo** not to output a new-line. This is often used in shell scripts to create input prompts.

C Shell

Type **echo -n "Press Return to continue:"** and press **Return**. The -n prevents echo from writing a new-line so the cursor stays at the end of the line.

---

4. Turn to Module 106 to continue the learning sequence.

Module 39
ed

DESCRIPTION

The external **ed** command is a line editor. It is the standard text editor for UNIX. A line editor displays a prompt while waiting for you to enter editing commands. It does not display a screen full of text for you to move the cursor around on and make changes. Instead, you display text with an edit command, then use other edit commands to change existing text or add new text.

You may wonder why anyone would still want to learn and use a line editor. There are a couple of reasons. The first is you may need a shell script to manipulate the contents of a file. Your choices are to use **ed**, **nawk**, or **sed**. But remember, **ed** is the base of the other two, and the benefit of using **ed** is you read and write in the same file. It takes time to read a file with **nawk**, redirect the output to another file, then move the file back to the original filename. The other reason for using **ed** is speed. It is fast compared to a full-screen editor when you need to perform simple known changes to text—especially on extremely large files.

COMMAND FORMAT Following is the general format of the **ed** command.

ed [**-p** *prompt*] [**-s**] [**-x**] [**-C**] *file*

red [**-p** *prompt*] [**-s**] [**-x**] [**-C**] *file*

BSD (Berkeley)

```
ed [ - ] [ -x ] file
```

Options The following list describes the options and their arguments that may be used to control how **ed** functions.

-C — Same as -x below except all text read into the buffer is assumed to be encrypted, unless a null key is specified.

-p *prompt* — The -p specifies that a user-defined prompt follows. This allows you to define your own for an internal edit command prompt. For example,

```
$ ed -p "infile: " infile
infile: _
```

causes **ed** to prompt you with *infile:* instead of the default : (colon) prompt.

-s — Suppresses the printing of information type messages. Character counts from the *e*, *r*, and *w* commands are suppressed. The diagnostics from the *e* and *q* commands are suppressed. The ! prompt displayed after a !*command* is also suppressed. On versions prior to SVR4 this was the - option.

-x An encrypted file is to be edited. The file is encrypted when it is saved to disk. It is decrypted when it is read into the editing buffer. You are required to enter an encryption key each time you start editing the file.

NOTE

The -x option is part of the Security Administration Utilities. These utilities are only available in the United States.

NOTE

The **ed** editor no longer supports the - option. The - option has been replaced with the -s option. System V only.

BSD (Berkeley)

- Same as System V -s option.
-x Same as System V -x option.

Arguments The following argument may be passed to the **ed** command.

file The name of the file you want to edit.

FUNCTIONAL OVERVIEW **ed** does not actually edit the specified *file*. It edits a *copy* of the file it has read into a temporary work file called a buffer. Thus all the edit commands you perform have no effect on the original file until you write the buffer back to the original file using the *w* command.

The **red** editor is a restricted version of **ed**. It only allows editing of files in the current directory. It does not allow you to escape to the shell from within the editor. If you do try to perform these types of commands, **red** responds with the message "restricted shell."

The tab formatting capability is supported by both **ed** and **red**. You must first perform a **stty -tabs** or a **stty tab3**, which sets your terminal mode for tab expansion. Then if the first line of the file being edited is a tab stop definition as defined by the **tabs** command (see Module 130), **ed** expands all tabs to the corresponding positions. For example, if the first line of your file is

```
:t4,8,12,16,20,24,28,32,36
```

tab stops are set at columns 4, 8, 12, and so on. Tabs embedded with the input of an *a*, *c*, and *i* command are not expanded according to the definition on the first line but, instead, to the default of every eight spaces.

MODES The **ed** editor has two modes: the input (insert) and command mode. In input mode the editor reads text (input) from your keyboard or a command file and inserts the text into the work buffer. In command mode the editor reads commanda (input) from your keyboard or a command file and performs the requested editing command. To enter insert mode you use one of the insert commands, *a*, *c*, or *i*. To exit insert mode and return to command mode you enter a . at the beginning of the line followed by a Return. When you enter the editor you are placed in command mode on the last line of the buffer.

REGULAR EXPRESSIONS The following table contains each regular expression and the task that it performs when used inside a pattern. A regular expression consists of regular alphanumeric characters matching themselves and special characters matching certain patterns of text. Regular

Expressions are often referred to as REs in UNIX terminology. Thus we use the RE notation for uniformity and briefness.

Alphanumeric RE	*Description*
c	Matches the character *c*.
string	Matches the set of characters *string*.

Metacharacters Metacharacters are the special characters used in regular expression patterns that have special meanings. Metacharacters are often referred to as *special* or *magic* characters or wild cards.

Special RE	*Description*
\	Escapes the meaning of a metacharacter.
^	Matches the beginning of the line.
$	Matches the end of the line.
.	Matches any single character.
[*class*]	A character class. Matches any one character in the class.
[*c1-c2*]	Match any one of the ASCII characters in the range defined within the brackets.
[^*class*]	Do NOT match any of the ASCII characters listed within the brackets. Ranges may be specified.
RE*	Matches zero or more occurrences of the preceding regular expression.
\(\)	Group the regular expression RE for later reference; each parenthetical RE is referenced by a corresponding number (see \n). The first group is referenced by \1, the second group by \2, and so on. You can have a maximum of nine groups in one regular expression pattern.
\n	Matches the *n*th grouped \(RE\) within the same regular expression.
RE\{*m*\}	Matches exactly *m* occurrences of the preceding one-character RE.
RE\{*m*,\}	Matches *m* or more occurrences of the preceding one-character RE.
RE\{*m*,n\}	Matches *m* through occurrences of the preceding one-character RE.
//	The last RE is used if it has been defined.

To have a metacharacter interpreted as a normal character precede it with a backslash (\).

ADDRESSING The **ex** editor has many ways of addressing lines of text within the current session buffer. The addresses should always precede the command they are affecting.

Basic Addressing The following table describes the basic addressing syntax available for your use.

Address Symbol	*Description*
	If no address is specified, the current line is affected or used for the address. Refers to the current line of the buffer. When you enter **ed**, the last line is the current line.
$	The last line of the buffer.

Address Symbol	*Description*
;	Refers to the current line to the end of the buffer (.,$).
>	
,	Refers to line one to the current line (1,.).
<	
n	Refers to line number *n*.
+	Refers to the next line.
+*n*	Refers to *n* lines forward.
^	Refers to the previous line.
-	
-*n*	Refers to *n* lines backward.
/*pat*/	Searches forward for the next line containing the string *pat*.
?*pat*?	Searches backward for the first previous line containing the string *pat*.
//	Repeats the last forward search.
??	Repeats the last backward search.
'*x*	Refers to line with mark *x*. The mark must be lowercase.
line	Any of the above basic addresses.

Address Ranges Some **ex** commands accept ranges of lines to work with. The following table describes the available ranges.

Address Symbol	*Description*
a1,*a2*	Specifies a range of lines beginning at line address *a1* and continuing through line address *a2*. Any of the basic addresses are valid. But *a1* must address a line that precedes address *a2*. For example, `/John/,/Mary/ d` deletes all lines from the line where the string *John* was found to the line where the string *Mary* was found. This assumes John resides on a line previous to the line where Mary resides.
a1;*a2*	Specifies a new current line and the number of lines to offset from the new current line. The address *a1* becomes your new current line and *a2* becomes the number of lines past the current line. For example, `10;5 co 50` copies line 10 through 15 to line 50. The 10 sets your current line to line 10 and the 5 includes the next 5 lines.
range	Any of the above addresses.

NOTE

If NO address is specified in an *ed* edit command, the current line is usually the default.

EDIT COMMANDS Each edit command you enter in **ed** must be followed by pressing Return. Some of the edit commands assume default addresses if you do not specify an address on the command line. These default addresses are shown in [ ] (brackets). The following table describes each of the edit commands.

Command	*Description*
[.]a *text*	The append command places you in insert mode. The text you type is entered into the buffer after the address you provide. If you do not provide an address, the current line is assumed. If you specify address 0, the text is inserted before the first line of the buffer. To exit insert mode and return to the command mode you must type a . (period) at the beginning of a line followed by a Return.
C	An encryption key is prompted for and is used to decrypt text read and written by all subsequent e, r, and w commands. An empty (null) key turns encryption off. See the -C option.
[.]c	The change command allows you to change existing text with new text. The text you enter replaces the text specified by the address. The new text you enter may be any number of lines. For example, if you specify two lines of text to change, you can immediately enter a . (period) and press Return. This would replace the two lines of text with no lines, the same as deleting them. Or you could enter 20 lines of text. It doesn't matter how many lines you specify in the address and how many new lines you enter. The current line becomes the last new line of text. If no new lines were entered, the current line becomes the previous line.
[.,.]d	The delete command deletes the specified lines from the buffer. The default address is the current line. The current line becomes the previous line.
e *file*	The edit command causes **ed** to edit a new file. The current contents of the buffer are deleted. The contents of the specified file is then read into the buffer. If you try using this command and **ed** responds with a ? then you have made modification to the buffer since the last save to disk. You can override the changes by typing *e* again. Your changes to the current buffer will be lost and the new file will be edited. You can save the changes by using the *w* command. Then re-enter the *e file* command. If no *file* is specified, the current filename is used. Thus you can re-edit the current file if you really foul things up in the buffer. Just type e Return and e Return again or use the E command.
E *file*	Same as the **e** command except **ed** does not check to see if you have outstanding modifications that need to be saved to disk. This command is useful if you decide to start over on the current editing session.
f *file*	The file command changes the current filename to *file*. If you do not specify a *file* then the current filename is displayed.
[1,$]g/*RE*/*ed-cmds*	The global command locates all of the lines that contain a string matching the specified regular expression *RE*. Each of the lines matched are set to the current line and the specified edit command *ed-cmd* is performed. For example,

```
5,25g/SVR4.0/d
```

Command	*Description*

	searches from line 5 to line 25 for the string "SVR4.0" and deletes any lines containing a match. If you do not specify an address, the entire buffer is searched. If you do not specify an *ed-cmd*, the *p* command is assumed. The *g*, *G*, *v*, and *V*, commands cannot be used as *ed-cmds*. Multiple *ed-cmds* may be specified by placing a \ (backslash) at the end of each line except for the last one. For instance,
	`g/UNIX/s//UNIX SVR3.2/\`
	`i\`
	`NOTE: A New Release of UNIX\`
	`.`
	locates each line containing "UNIX." A substitution replaces "UNIX" with "UNIX SVR3.2" and a note is inserted before each of these lines.

[1,$]G/*RE*/	This is the global interactive command. It locates all lines that match the given regular expression *RE*. Each of the lines matched is set to the current line. The editor displays each line and you are required to enter one and only one *ed* command. You cannot use the *a*, *c*, *i*, *g*, *G*, *v*, or *V* command. The command you specify is performed on the current line. After the command is completed the next line containing a match is displayed. To have no command executed on the current line press Return. You can repeat the last command by typing the & (ampersand) key and pressing Return. To abort the interactive global command press the Delete or Break key.
h	Displays a brief error message explaining the cause of the last ? diagnostic message. If something is wrong with a command, **ed** displays a ? (question mark). You must type h and press Return to get an explanation.
H	Turns the help feature on and off. If the help feature is on, then a brief message is displayed instead of the ?. If the help feature is off, the ? is displayed and you must use the h command to display the brief explanation.
[.]i	The insert command is like the *a* (append) command, except *test* the text is inserted before the addressed line instead of after it. You cannot use the *i* command in an empty file. Nor can you specify address 0. The last inserted line becomes the new current line when you return to the command mode. If no new text is inserted, then the address specified becomes the new current line.
[.,+1]j	The join command removes the new-line character at the end of each line. Thus the addressed lines become one long line.
[.]k*c*	The mark command marks the addressed line with letter *c*. The letter *c* must be lowercase. You use the k command to set new addresses in the buffer. Then you can reference the address with the '*c* address convention. The *c* may be any character from a to z.
[.,.]l	The list command displays the nonprintable characters in a visual representation. Control characters are displayed as ^*c* or as octal numbers, depending on the function they perform. You can use the *l* command with any other edit command except the *e*, *f*, *r*, or *w* command.

Command	*Description*
[.,.]m*adr*	The move command moves the addressed lines to reside after the line specified by *adr*. You can specify an address of 0 to move the lines to the beginning of the buffer. The address *adr* cannot be within the specified range of lines being moved. The current line becomes the last line moved.
[.,.]n	The number command displays each addressed line preceded by a number and a tab. The numbers are not part of the buffer contents; they are for your information only. The current line becomes the last line displayed. You can use the *n* command in conjunction with any command except the *e*, *f*, *r*, or *w* command. If no address is given, the current line is displayed.
[.,.]p	The print command displays each addressed line. The current line becomes the last line displayed. You can use the *p* command in conjunction with any command except the *e*, *f*, *r*, or *w* command. If no address is given the current line is displayed.
P	The prompt command turns the default prompt on or off. The default prompt is an * (asterisk). The prompt is initially turned off.
q	The quit command causes **ed** to exit. If you have saved the contents of the buffer to disk since the last modification, **ed** exits. If you have outstanding modifications, **ed** returns a ? or "file not saved" message. You can use the *w* command to save the buffer contents to disk, then retype *q*. If you do not want to save the changes to disk, you can repeat the *q* command again and **ed** will exit.
Q	The nonchecking quit command. This command is equivalent to typing two *q* commands. The editor does not check for unsaved changes. It immediately exits the buffer whether you have saved modifications to disk or not.
[$]r *file*	The read command reads in the specified *file* and places the contents after the addressed line. By default, the *file* is appended to the end of the buffer. If you specify address 0, the *file* is inserted at the beginning of the buffer. If the *file* is read in without error, the number of bytes read in are displayed. If you do not specify a *file*, the currently remembered filename is used. The *r* command does not reset the remembered filename unless it is the first filename used in the current editing session.
[$]r !*command*	The read command can also read the standard output of any UNIX command. This is like the *r file* command except the input is read from a command instead of a file. For example, `r!ls` reads the output of the **ls** command executed in the current directory.
[.,.]s/*RE*/*rstr*/ [.,.]s/*RE*/*rstr*/g [.,.]s/*RE*/*rstr*/*n*	The substitute command searches each specified line for a matching regular expression *RE*. If a match occurs, the first *RE* on the line is replaced with the specified *rstr* string. If you use the *g* suffix, all occurrences of *RE* are replaced on each line (global line substitute). The *n* suffix causes only the *n*th occurrence of *RE* to be replaced on each line. The last line a substitution occurs on becomes the new current line.

Command	*Description*
	The delimiter character may be any character except a space or new-line. For example, instead of s/old/new/ you can use s#old#new#.
	The & character may be used in the *RE* or *rstr* (replacement string) to represent the last *RE*. For example, `s/UNIX/&-TM/` substitutes UNIX to be UNIX-TM.
	The % character may be used in the *RE* or *rstr* to represent the last *rstr*. For example, `s/OS/%/` substitutes OS to be UNIX-TM, assuming you just performed the previous example s/UNIX/&-TM/.
[.,.]t*adr*	The copy command works like the move command (m) except a copy of the addressed text is placed after the *adr* address. The current line becomes the last line of the copied text.
u	The undo command reverses the effects of the last text manipulation command.
[1,$]v/*RE*/ed-cmds	The global exception command functions like the *g* command except it selects lines that do not match the specified *RE*. For example, `v/^$/p` displays all lines in the buffer that are not empty lines. The ^$ regular expression means beginning of line followed by end of line, which is equal to an empty line. Refer to the *g* command for further information.
[1,$]V/*RE*/	Like the interactive *G* command, except it selects lines that do not match the specified *RE*. Refer to the *G* command for further details.
[1,$]w *file*	The write command writes the specified lines from the buffer to the *file*. If the write is successful, the editor returns the number of bytes (characters) written to the file. If you do not specify an address range, the entire buffer is saved to disk. If you do not specify a file, the current remembered filename is used. This command does not change the remembered filename unless the specified filename happens to be the first filename used in the editor.
[1,$]W *file*	The same as the *w* command except the lines are appended to the specified *file*.
[1,$]w!*command*	The same as the *w* command except the lines are written to the standard input of the specified UNIX *command*. For example, `10,50w!pr -t2` writes lines 10 through 50 to the UNIX **pr** command. The **pr** command formats the input into two-column output.
X	Same as -C but an educated guess is made as to whether or not the text read (in by e and r commands) is encrypted.

Command	*Description*
[$]=	The line number command displays the line number of the specified address. If no address is given, the last line is used, thereby giving you the total number of lines in the file. Another common use is the .= format which returns the current line number.
!*command*	The escape command escapes from the editor and invokes the shell to execute the specified UNIX *command*. For example, `!grep mylogin /etc/passwd` greps for the string *mylogin* from the passwd file. This command does not read the output of the *command* into the buffer. To do that you must use the r! command. The % character may be used in the *command* to represent the currently remembered filename. For example, `!%` executes the current file you are editing. This is useful if you are editing a shell script and need to check it before exiting the editor.
Address Commands	
<CR>	Pressing Return on a line by itself displays the next line and increments the current line number.
n	A number command may be used to change the current line position to the given line number.
.	The dot represents the current line and thus displays the current line when used by itself.
; >	The ; and > characters represent addresses .,$, which is from the current line to the end of the buffer.
, <	The , and < characters represent addresses 1,. which is from the first line to the current line.
&	The repeat command repeats the last command you entered.

In most cases you can only specify one command per line. The *l*, *n*, and *p* commands are exceptions to this rule. You can specify these three commands with other commands. This is because they are all display oriented commands. But you cannot use them with the *e*, *E*, *f*, or *w* commands.

DIAGNOSTICS AND BUGS **ed** has very brief and usually useless error messages. Fortunately, there usually isn't too much to explain with a line editor.

?	Indicates that you made a mistake on the edit command. Usually, you have a syntax problem. Check the edit command format and try again. Turn on the help feature by typing H and pressing Return.
?*file*	Indicates that **ed** could not read or write to the specified file. You might receive this message when using the *e*, *r*, and *w* commands. It usually means the file exists and you do not have read or write permissions (whatever the case may

be). It may mean you do not have read or write permissions for the directory. In some instances it could also mean the file system is out of disk space. Try the !df command to verify disk space is available.

RELATED COMMANDS Refer to the **nawk, ex, sed,** and **vi** commands described in modules 6, 43, 117, and 151.

RELATED FILES The editor uses several files and directories while you are editing.

/var/tmp	The directory where the temporary file buffer is stored.
TMPDIR	The shell variable containing the name of the directory to use instead of the default /var/tmp directory. If the variable is set, the specified path is used; otherwise, /var/tmp is used.
ed.hup	The contents of the buffer are saved to this file if the line hangs up (you get disconnected from the system). This is usually caused by a phone line or modem problem, although even direct cabling has been know to be pulled loose from the computer or terminal by clumsy people.

APPLICATIONS

The **ed** editor can be used to create new text files or update existing ones. It is a fairly powerful line editor. Although most computer users have moved to screen editors, there is still a need for line editors. The most common use is in shell scripts. It is often necessary to make changes to a file from within a shell script. The easiest way to do this may be by using a line editor like **ed**. One thing that sets UNIX editors apart from most editors is the rich set of wild cards, called Regular Expressions on UNIX. Regular Expressions allow you to match almost any existing string of text you can imagine.

You may find the need to use a line editor when you cannot figure out what the proper terminal type is for a certain terminal you are using. Until you find out the terminal type you cannot use the **vi** screen editor. Therefore, it is a good idea to know a little about the line editors just in case you get in a time squeeze and are forced to do some editing before you can use **vi**.

TYPICAL OPERATION

In this activity you use the **ed** command to create and modify a text file. Begin at the shell prompt.

1. Type **ed ed_file** and press **Return**. Notice the following display:

```
cj> ed ed_file
?ed_file
_
```

   Notice there is no internal prompt.

2. Type **a** (for append) and press **Return** to begin entering text. Notice the cursor moves to the next line.

```
cj> ed ed_file
?ed_file
a
```

   Again notice no prompt during insert mode either.

3. Type the following text, pressing **Return** at the end of each line.

```
cj> ed ed_file
?ed_file
a
This is a new file created by the ed editor.
It is a line editor.
I prefer the screen editor.
But I need to know how to use this editor in certain situations.
Time to return to the command mode.
```

4. Type a **.** (period) at the beginning of the next line to return to the command mode.
5. Type **w** and press **Return** to save the contents of the buffer to disk and exit the editor.
6. Type **q** and press **Return** to save the contents of the buffer to disk and exit the editor.
7. Type **ed -p "ed " ed_file** and press **Return** to re-edit the *ed_file*. Notice the following display:

```
cj> ed -p "ed " ed_file
195
ed>
```

   The 195 is the number of characters **ed** read from the *ed_file* file.

8. Display the entire contents of the buffer by typing **,p** and pressing **Return**.
9. Display the current line by typing **p** and pressing **Return**. The current line is the last line in the buffer.
10. Delete line 2 by typing **2d** and pressing **Return**.
11. Insert a new line before the last line of the file as follows:
    a. Type **$i** and press **Return**.
    b. Type the line **This is a newly inserted line of text.** and press **Return**.
    c. Type **.** and press **Return** to return to command mode.
12. Display the contents of the buffer by typing **,p** and pressing **Return**. Notice the changes to the buffer.

```
ed> ,p
This is a new file created by the ed editor.
I prefer the screen editor.
But I need to know how to use this editor in certain situations.
This is a newly inserted line of text.
Time to return to the command mode.
```

13. Change the word "new" on line 1 to "existing" by typing **1s/ new/n existing/p** and pressing **Return**.

```
ed> 1s/ new/n existing/p
This is an existing file created by the ed editor.
```

   The space before the word "new" is replaced by the character "n" followed by a space. This corrects the preceding "a" to be "an."

14. Move lines three and four to the beginning of the file. Type **3,4m0** and press **Return**.

15. Type **,p** and press **Return** to display the new changes.

```
ed> ,p
But I need to know how to use this editor in certain situations.
This is a newly inserted line of text.
This is an existing file created by the ed editor.
I prefer the screen editor.
Time to return to the command mode.
```

16. To save the changes to disk type **w** and press **Return**.
17. To exit the editor type **q** and press **Return**.
18. To remove unneeded files and conserve disk space remove the *ed_file* by typing **rm ed_file** and pressing **Return**.
19. Turn to Module 43 to continue the learning sequence.

Module 40
egrep

DESCRIPTION

The external **egrep** command is used to search ASCII text files for a given text pattern. All lines containing the pattern are written to the standard output. The pattern may be a word or string of characters. Full regular expressions may be used inside the pattern to specify certain sequences of characters. If **egrep** searches more than one file, each displayed line is preceded by the filename. If no filenames are given, **egrep** reads from the standard input. Thus you can use **egrep** in a pipe command.

The name **egrep** is a combination of editor command characters. It is from the editor command :g/RE/p, which translates to global Regular Expression print. Inside the editor this command would search the entire file for all lines matching the RE *pattern*. The -v option of **egrep** is from the editor command :v/RE/p, which searches the entire file for all lines except the ones containing the RE *pattern*. The e added to the front of **egrep** stands for expression grep. That is a grep that can search for full regular expression type strings.

The **egrep** command adds the feature options of the **fgrep** command and extends the pattern matching capabilities of the **grep** command. You can use full regular expressions to search for matches. These expressions can be stored in a file or used on the command line. Thus **egrep** is the most powerful of the grep commands but also requires the most system resources and runs the slowest in most cases. For flexibility and pattern matching power you lose speed.

NOTE

The **egrep** command performs line oriented searches. Therefore, a phrase that spans more than one line is not matched. When deciding what string to use for the search, you should try to keep it as short as possible yet as unique as possible.

COMMAND FORMAT Following is the general format of the **egrep** command.

```
egrep [ -bchilnv ] [ -e pattern ] pattern file_list
egrep [ -bchilnv ] [ -e pattern ] -f file file_list
```

BSD (Berkeley)

```
egrep [ -bclnsv ] [ -e pattern ] pattern file_list
egrep [ -bclnsv ] [ -e pattern ] -f file file_list
```

The form of the command uses the *pattern* provided on the command line to search the *file_list* for a match. In the second form multiple *patterns* are stored in *file* and **egrep** searches the file_list for each *pattern* listed.

Options The following list describes the options and their arguments that may be used to control how **egrep** functions.

-b	Displays the block number in which the *pattern* was found before the line that contains the matching *pattern*.
-c	Displays only a total count of matching lines for each file processed.
-e *-string*	This allows you to specify a string that begins with a dash. Normally, any argument beginning with a dash is interpreted as an option, not a string or argument.
-f *file*	Read in the strings to search for from *file*. This allows you to create a file containing all of the strings you want **egrep** to search for in the file_list or standard input.
-h	Suppress the displaying of filenames which precede lines that match the specified patterns when multiple files are searched.
-i	Ignore the difference between uppercase and lowercase characters during comparisons.
-l	Displays only the names of the files containing the specified *pattern*. The lines containing the *patterns* are not displayed.
-n	Displays the line number before each line containing the *pattern*.
-v	Displays only the lines that do not match the *pattern*. The v command in the **ex** editor performs the same type of function. It is an exception search. Search for every line except the ones containing the given *pattern*.

BSD (Berkeley)

-h	Not supported.
-i	Not supported.
-s	Suppresses the displaying of diagnostic error messages for nonexistent and nonreadable files.

Arguments The following list describes the arguments that may be passed to the **egrep** command.

pattern	Any combination of characters, numbers, and regular expression patterns. **egrep** searches for strings of text that match the *pattern* expression.
file_list	One or more files to search for the given *pattern*.

SEARCH PATTERNS (REGULAR EXPRESSIONS) The following table contains each regular expression and the task that it performs when used inside a *pattern*. A regular expression consists of regular alphanumeric characters matching themselves and special characters matching certain patterns of text. Regular Expressions are often referred to as REs in UNIX terminology. Thus we use the RE notation for uniformity and brevity.

Alphanumeric RE	*Description*
c	Matches the character *c*.
string	Matches the set of characters *string*.

Metacharacters Metacharacters are the special characters used in regular expression patterns that have special meanings. Metacharacters are often referred to as *special* or *magic* characters or wild cards.

Special RE	*Description*
\	Escapes the meaning of a metacharacter.
^	Matches the beginning of the line.
$	Matches the end of the line.
.	Matches any single character.
[*class*]	A character class. Matches any one character in the class.
[*c1-c2*]	Match any one of the ASCII characters in the range defined within the brackets.
[^*class*]	Do NOT match any of the ASCII characters listed within the brackets. Ranges may be specified.
|	Alternation of regular expressions. Matches either one or the other of the regular expressions provided.
( )	Concatenation operation. Normally the parentheses are omitted. Allows for control of precedence in the interpretation of the regular expressions specified.
*	Matches zero or more occurrences of the preceding regular expression.
+	Matches one or more occurrences of the preceding regular expression.
RE?	Matches zero or one occurrences of the preceding regular expression.
RE\{*m*\}	Matches exactly *m* occurrences of the preceding one-character RE.
RE\{*m*,\}	Matches *m* or more occurrences of the preceding one-character RE.
RE\{*m*,n\}	Matches *m* through *n* occurrences of the preceding one-character RE.

To have a metacharacter interpreted as a normal character, precede it with a backslash (\).

DIAGNOSTICS AND BUGS The input lines read by **egrep** are limited to the system's BUFSIZ definition. If a line is longer than BUFSIZ characters, the line is truncated before the comparison is made. The BUFSIZ used by your system can be found in the /usr/include/stdio.h file.

The file containing regular expressions must not contain a blank line. If it does contain a blank line, **egrep** complains with a syntax error.

RELATED COMMANDS Refer to the **nawk**, **ed**, **ex**, **fgrep**, **grep**, **sed**, and **ksh** commands described in modules 6, 39, 43, 52, 60, 117, and 71.

RELATED FILES The **egrep** command can read from the standard input or a specified list of files. It writes to the standard output.

RETURN CODES The return code is set to 0 if any *patterns* were found. If no *patterns* were found, the return code is 1. If syntax errors are present in the *pattern* or if files are not accessible, a return code of 2 is returned, even if *patterns* were found.

APPLICATIONS

You can use **egrep** to search text for a specific word or string of characters. The text may be contained in a file or be from the standard input. The various uses of grep include finding lines or files that contain a specific *pattern*, counting the number of lines that contain the *pattern* in a file, showing the line numbers of lines that contain the *pattern*, and showing the block that contains the *pattern*. These uses may be inverted to function on all lines that do not contain the specified *pattern* also.

The uses of **egrep** are far reaching, from just searching a file for lines that contain a *pattern* to using **egrep** inside shell scripts to verify a *pattern* exists. It is one of those commands that doesn't seem special until you use it a few times, then you can't seem to live without it.

Because **egrep** is a combination of the **fgrep** and **grep** commands, some users choose to use only the egrep command. There is no real problem with this practice except for the speed of **egrep** compared to **grep** and **fgrep**. The decision is yours and depends on the requirements that you have each time you use one of the commands.

TYPICAL OPERATION

In this activity you use the **egrep** command to locate your login in the passwd file. Various *pattern* examples are also used to show you how to use the regular expressions. Begin at the shell prompt.

1. Type **egrep "^$LOGNAME" /etc/passwd** and press **Return**.

C Shell

If you are using the **csh**, replace $LOGNAME with $USER.

---

2. Type **egrep "$HOME|HOME" .profile** and press **Return** to search for the string "/u1/ts/mylogin" ($HOME) and the string HOME in your .profile file. The $HOME variable is expanded on the command line to be your home directory path. The HOME string is taken as a literal string.

C Shell

If you are using the **csh**, use the *.cshrc* and *.login* filenames in place of *.profile*.

---

3. Type **who | egrep "bill|nancy"** and press **Return**. You may have to change the "bill" and "nancy" patterns to a different pattern, but this command returns if certain users are logged in on the system.
4. Turn to Module 52 to continue the learning sequence.

Module 41
env (SV)

DESCRIPTION

The external **env** command creates a new environment for executing a given command. Depending on the options, **env** either uses the current environment or completely ignores it. If **env** uses the environment, modifications are made based on the arguments given. After the new environment is set up, the command argument is executed.

COMMAND FORMAT Following is the general format of the *env* command.

```
env [ - ] [ name=value ... ] [ command [ options ] [ args ] ]
```

Options The only valid option for **env** is described as follows:

- Do not pass the current environment to the command being executed by **env**.

Arguments The following list describes the arguments that may be passed to the **env** command.

name=value Variables may be assigned values that will be exported for the command to use during execution.

command The command that **env** executes with the new environment. The options and arguments are those of the command being executed; they are not interpreted by **env**.

If no arguments are specified, **env** displays all exported variables.

FURTHER DISCUSSION Arguments must be of the form *variable=value*. These arguments are merged into the existing environment. The remainder of the arguments are executed as a normal command. The command executes with the newly defined environment. After the command exits, the environment returns to the state it was before the **env** command was executed.

The following command shows the changes made to the environment by **env**:

```
cj> env HOME=/tmp TERM=adm5 env
ENV=/u1/ts/mylogin/.kshrc
FCEDIT=vi
HOME=/tmp
LOGNAME=mylogin
MAIL=/usr/mail/mylogin
PATH=:/u1/ts/mylogin/bin:/bin:/usr/bin:/usr/lbin:/usr/local/bin
SHELL=/bin/ksh
TERM=adm5
TMPDIR=/tmp
TZ=CST6CDT
VISUAL=vi
```

The **env** command modifies your HOME and TERM variables and then executes the **env** command again to show the changes in the new temporary environment.

If you do not wish to have the current environment passed to the command for execution, you can use the - (hyphen) option. Only the variables you define on the **env** command line will be passed to the command. The following command demonstrates the use of the dash option:

```
cj> env - HOME=/tmp TERM=adm5 env
HOME=/tmp
TERM=adm5
```

The **env** command will create a new temporary environment with only the HOME and TERM variables defined.

If no arguments are given, **env** lists out the current environment with one variable and value per line.

RELATED COMMANDS Refer to the **ksh** command described in Module 71.

RELATED FILES The **env** command writes to the standard output.

APPLICATIONS

The **env** command is useful for testing commands in controlled environments. It may be used to execute a command in a new environment without affecting your own personal environment. It is probably used more to display the current environment than as a command executor.

TYPICAL OPERATION

In this activity you use the **env** command to display your current environment. Begin at the shell prompt.

1. Type **env** and press **Return** to list out your current environment.

```
cj> env
ENV=/u1/ts/mylogin/.kshrc
FCEDIT=vi
HOME=/u1/ts/mylogin
LOGNAME=mylogin
MAIL=/usr/mail/mylogin
PATH=:/u1/ts/mylogin/bin:/bin:/usr/bin:/usr/lbin:/usr/local/bin
SHELL=/bin/ksh
TERM=vt100
TMPDIR=/tmp
TZ=CST6CDT
VISUAL=vi
```

2. Turn to Module 118 to continue the learning sequence.

Module 42
eval

DESCRIPTION

The internal **eval** command interprets and expands a command line before the shell interprets and expands the line. This allows you to:

- Execute a command line you read using the *read* command.
- Find the value of a variable whose name is derived from the value of another variable.
- To repeat a stage or portion of command processing.
- To apply results of expansions to earlier stages of the expansion.

COMMAND FORMAT Following is the general format of the **eval** command.

```
eval [ command arg ... ]
```

Arguments The following list describes the arguments that may be passed to the **eval** command.

command The name of any valid command (internal or external).

arg The arguments used by the specified command. The *arg*s are separated by a space.

EVALUATION The shell expands arguments to **eval** using standard command processing rules. Then the shell forms a space-separated string of all the arguments. The shell reads the string as a command line and processes it again and executes it.

You can use **eval** to execute a command you read using the *read* command. For example,

```
cj> read CMD
cat /etc/group | grep mylogin
cj> eval "$CMD"
```

processes the CMD variable. If you enter any special characters, such as ; or |, eval expands them and the shell reprocesses the results before executing the command line. If you did not use the **eval** command, the CMD variable would not be interpreted correctly and certain commands would not run. Another example illustrates this concept.

```
cj> CMD='date | wc'
cj> $CMD
usage: date [-a sss.fff] [-u] [tformat] [mmddhhmm[yy]]
cj> eval $CMD
Sat Jan 5 22:54:04 CST 1989
```

The first line sets CMD to a command containing a pipe. The second command trys to execute the variable CMD. The execution fails because the pipe is not expanded and is passed to **date** as an argument. Thus **date** complains about a bad argument. The third line expands CMD to **date | wc**. The shell executes the line as a proper command.

The following example illustrates how you can use **eval** to expand variables within a command line to be the name of another variable.

```
cj> eval last='$'{$#}        # expands to last positional parameter
cj> X=10
cj> Y=X
cj> echo '$'$Y
$X
cj> eval echo '$'$Y
10
```

APPLICATIONS

The **eval** command provides an extended feature allowing you to do special command processing. It is used to evaluate variables twice, so you can use the value of one variable as a variable name. The other popular use of eval is to process an input line as a command. Without **eval** this would be next to impossible.

TYPICAL OPERATION

In this activity you use the **eval** command to expand the positional parameters to check for a file name being specified in the last parameter. Begin at the shell prompt.

1. Type the following lines of code; press **Return** at the end of each line.

```
cj> set One Two Three Four
cj> if eval [ ! -f \${$#} ]
then echo "$4: last argument must be a file!"
     #exit 1
fi
```

C Shell

There is no reason for eval to perform this function in the **csh** since the **$#**argv option is available. So we'll show you a different use for eval also useful in **sh** and **ksh**. This use of eval is often seen in shell scripts.

```
cj> set input=$<
$TERM
cj> echo $input
$TERM
cj> eval echo $input
vt100
```

This code checks for the last argument on the command line to be the name of a file that exists. The first line sets the positional parameters to One Two Three Four. Since Four is not the name of a file, your code will return the message about the last file. The exit is commented out so you are not logged off when the command executes.

2. Turn to Module 44 to continue the learning sequence.

Module 43
ex

DESCRIPTION

The external **ex** command is a powerful line editor that forms the base of the **vi** editor. It is a supet-set of the original UNIX line editor **ed**. The basic line editing commands, macros, abbreviations, tags, substitutions, options, the EXINIT variable, and the .exrc file are discussed in this module. The visually oriented aspects of **ex/vi** are discussed in Module 151.

COMMAND FORMAT Following is the general format of the **ex** command.

```
ex [ -CRLsvx ] [ -r lost ] [ -t tag ] [ -c cmd ] [ file ]
```

BSD (Berkeley)

```
ex [ - ] [ -lv ] [ -r lost ] [ -t tag ] [ +cmd ] [ file ]
```

Options The following list describes the options and their arguments that may be used to control how **ex** functions.

-C	Same as -x below except all text read into the buffer is assumed to be encrypted, unless a null key is specified.
-R	Read-only mode; no updates allowed.
-L	List the names of all files saved when the system crashed or the editor aborted.
-c *cmd*	Display buffer after interpreting an **ex** command *cmd*. Enclose commands with special characters in single quotes ('*cmd*') to escape shell interpretation.
-r [*lost*]	Recover *lost* file that was preserved during abnormal termination of **ex/vi**. Use -r option alone for list of preserved files.
-s	Suppress all standard output (interactive feedback). Used for batch editor script processing.
-t *tag*	Starting at *tag*, edit the file that is referenced by *tag* in the "tags" file.
-x	Edit encrypted files. You are prompted for an encryption key.
-v	Invoke **vi**.

BSD (Berkeley)

-	Same as System V -s option.
-l	Sets the showmatch and lisp mode options. The ( ), { }, [[and]] commands are changed to indent for lisp code.
+cmd	Same as System V -c cmd. Plus (+) with no *cmd* begins editing at last line of buffer.

Arguments The following argument may be passed to the **ex** command.

file_list One or more files to edit. If you specify more than one file, **ex** edits the first file on the command line. To edit the next file, you enter :n and press Return.

Table of Contents The following is a brief table of contents that will help you locate information in this module.

FUNCTIONAL OVERVIEW 170

ACCESS TO VI 170
vi Mode 171

ADDRESSING 171
Basic Addressing 171
Address Ranges 172

SPECIAL ex CHARACTERS 172

COMMANDS 173
Command Format 174
Displaying Text 174
Window Sizing 174
Inserting Text 175
Changing Text 175
Copying Text 175
Deleting Text 176
Marking Positions 176
Moving Text 176
Moving Text (cut & paste) 176
Yanking (cutting) 176
Putting (pasting) 177
Moving text between files 177
Indenting Text 177
Filtering Text 177
Join Lines 178
Undo, Redo, and Retrieve 178
Edit/Re-edit a File 178
Edit/Re-edit Multiple Files 179
Read from a File 180
Write to a File 180
Escape to Unix 180
Searches 181

Global Searches 181
Global Exception Search 182
Substitutions 182
Substitute Options 182
REGULAR EXPRESSIONS 183
Metacharacters 183
Special Substitute Characters 183
TAGS 184
Tag File Format 185
Referencing a Tag 185
MACROS 185
MISCELLANEOUS 185
LEAVING ex 186
CUSTOMIZING OPTIONS 186
The Options 187
The .exrc File 190
The EXINIT Variable 191

FUNCTIONAL OVERVIEW The **ex** editor is a line editor. Most of the commands are line oriented. Thus to perform character changes and deletions you have to use the substitute command or open mode. Open mode is visual editing on one line of text. You may find these methods to be tedious and not very productive, which is why **vi** exists.

It is also necessary to provide unique amounts of each command's name. This causes the names in **ex** to have more meaning than the single keystrokes used in **vi** but may cause some confusion between the two editors. A suggestion is to utilize **vi** for all character and line oriented editing and the moving of text with operators. While using **ex** to perform file operations, substitutions, setting macros, setting abbreviations, and tags. Obviously, there is overlapping of functions but in general, **ex** supplies a gateway to administrative commands used while in **vi**.

NOTE

Each command in **ex** command mode must be followed by a Return. When you are in **vi** command mode, the characters are interpreted as they are typed and no Return is needed in most cases.

Interface capabilities The **ex** editor can interface with **vi** mode to provide full-screen editing. It has the ability to interface to the UNIX system. A subshell may be entered from **ex** and UNIX commands executed until a Ctrl-D or exit is entered. UNIX commands can be executed from **ex** without leaving the editor. The **vi** mode can interface with UNIX filters to allow for special editing capabilities.

ACCESS TO VI You can move between the **ex** and **vi** editors with simple commands. The **vi** editor is the visual extension of the **ex** editor. To access **vi** from **ex** you simply type **vi** at the colon prompt. You must have your TERM variable set correctly for **vi** to function properly. **ex** also requires limited knowledge of your terminal type for certain commands and features. Refer to the sections in the **vi** module that discuss the requirements for setting up your environment to use **vi**.

vi Mode The following commands may be used to interface with the **vi** screen editor and return to the **ex** screen editor.

:vi	Enter/Return to **vi** mode. If you are in **ex** you can enter or return to **vi**.
:open	Enter into *open* mode, a pseudo screen control mode entered from **ex**. Makes it easier to view text. In open mode you can make visual changes to individual lines. You're halfway to **vi**.
Q q	Enter/Return to **ex** mode, return to **vi** mode with :vi. You may want to use **ex** if you have a lot of substitutions to perform, or if you entered **vi** with the wrong TERM definition. You can enter **ex** mode and type **set term=*termtype*** and then type **vi** to return to vi.
:	Enter temporary **ex** mode while in **vi**. Precede each **ex** command you need to perform with a : (colon). This causes **vi** to send the command to **ex** for processing.

ADDRESSING The **ex** editor has many ways of addressing lines of text within the current session buffer. The addresses should always precede the command they are affecting.

Basic Addressing The following table describes the basic addressing syntax available for your use.

Address Symbol	*Description*
	If no address is specified the current line is affected or used for the address.
.	Refers to the current line of the buffer. When you enter **ex** the last line is the current line.
$	The last line of the buffer.
%	The entire buffer (lines 1,$).
n	Refers to line number *n*.
+	Refers to the next line.
+*n*	Refers to *n* lines forward.
-	Refers to the previous line.
-*n*	Refers to *n* lines backward.
/*pat*/	Searches forward for the next line containing the string *pat*.
?*pat*?	Searches backward for the first previous line containing the string *pat*.
//	Repeats the last forward search.
??	Repeats the last backward search.
'*x*	Jumps to the line with mark *x*.
''	Jumps to the line that was current before the last nonrelative motion.
line	Any of the above basic addresses.

Address Ranges Some **ex** commands accept ranges of lines. The following table describes the available ranges.

Address Symbol	*Description*
a1,a2	Specifies a range of lines beginning at line address *a1* and continuing through line address *a2*. Any of the basic addresses are valid. But *a1* must address a line that precedes address *a2*. For example, `/John/,/Mary/ d` deletes all lines from the line where the string *John* was found to the line where the string *Mary* was found. This assumes John resides on a line previous to the line where Mary resides.
a1;a2	Specifies a new current line and the number of lines to offset from the new current line. The address *a1* becomes your new current line and *a2* becomes the number of lines past the current line. For example, `10;5 co 50` copies line 10 through 15 to line 50. The 10 sets your current line to line 10; the 5 includes the next 5 lines.
range	Any of the above addresses.

NOTE

If NO address is specified in an **ex** command, the current line is usually the default.

SPECIAL ex CHARACTERS These characters have special meaning to the **ex** editor when you are at the colon (:) command prompt.

%	Name of current buffer/file. Address for entire buffer, lines 1 through end-of-file.
#	Name of alternate buffer/file, last file edited or used in **ex** command. Is also a command alias for the *nu*mber command.
!	Ignore changes when used with :e, :w, :n. If you have modified the contents of the buffer, **ex** will normally not allow you to perform these functions without saving the buffer to disk. The exclamation mark (!) overrides this feature. Also used as a temporary escape to UNIX to execute a single UNIX command line.
!!	Repeats the last escape to UNIX command.
|	Sequential **ex** command separator. You can place multiple **ex** commands on the command line and separate them with a vertical bar.
;	Sequential **ex** command separator for tag file use. If you want to have multiple **ex** commands performed for a specific tag, you can separate the **ex** commands with the semicolon in your tag file.

COMMANDS The following table lists all valid **ex** commands, their abbreviated form, and a brief description.

Command	*Abbr.*	*Description*
abbrev	ab	Used to define abbreviation strings.
append	a	Enter insert mode after current line.
args	ar	List args supplied on **ex/vi** command line.
change	c	Change the specified lines.
copy	co	Copy the specified lines.
delete	d	Delete the specified lines.
edit	e	Edit the specified file.
file	f	Display status of current file/buffer.
global	g	Address entire buffer.
insert	i	Enter insert mode.
join	j	Join the next line to the current line.
list	l	List all characters in raw format. Shows control characters.
map		Map a character to perform **ex/vi** commands. If followed by a ! (map!) then use as create a substitute string used during insert mode.
mark	ma	Mark current line for later reference.
move	m	Move to specified line.
next	n	Edit the next file on the command line.
number	nu	Number each line when displayed.
preserve	pre	Preserve a copy of the current buffer.
print	p	Print (display) lines of text.
put	pu	Put text in buffer from a register.
quit	q	Quit the editor without saving changes.
read	re	Read in text from a file or UNIX command.
recover	rec	Recover a lost (preserved) file.
rewind	rew	Rewind to the beginning of the files on the command line.
set	se	Set an **ex** option.
shell	sh	Escape to a UNIX shell.
source	sop	Read in and interpret a .exrc type file.
stop	st	Suspend (Stop) the editor. (Job Control Systems only)
substitute	s	Substitute an old string with a new string.
unabbrev	una	Remove an abbreviation string.
undo	u	Undo the last command.
unmap	unm	Unmap the specified map command.
version	ve	Display the version **ex/vi** you are using.
visual	vi	Enter into visual mode, **vi**.
write	w	Write the current buffer to a disk file.
xit	x	Exit the editor and save any changes to disk.
yank	ya	Yank specified lines to a register.
window	z	Set window to the specified size.
escape	!	Escape to UNIX to execute a command.
lshift	<	Shift text one shiftwidth to the left.

Command	*Abbr.*	*Description*
print next	Return	Display the next line of text.
resubst	&	Repeat the last substitute command.
rshift	>	Shift text one shiftwidth to the right.
scroll	Ctrl-D	Scroll the screen down one page.

Command Format The **ex** commands have a general format of

```
address command count flags
```

where address, count, and flags are optional. The address is a *range* or *line* as described in the Address section. If no *address* is specified, the current line is usually the default. The command is one of the commands listed in the previous table. The *count* specifies how many lines are affected by the command. If you specify 3d4, the current line is set to line 3 and 4 lines are deleted. The *flags* can be one of the following *commands*.

l	List the lines, displaying control characters with visual characters.
nu #	Display the lines preceded by line numbers.
p	Display (print) the lines.
+*n*	Set the current line forward *n* lines.
-*n*	Set the current line backward *n* lines.

Displaying Text The following commands may be used to display the contents of the current session buffer you are editing.

range p *count* — Print (display) lines specified by *range* or perform p *count* times from specified line.

range nu *count flags* — Print (display) lines specified by *range*, preceded by line numbers. Flags may be used to perform added functions. For example,

```
:%nu l
```

displays all lines preceded by the line number, with control characters displayed.

range l *count flags* — List lines specified by *range*, showing nonprintable characters.

Ctrl-D — Scroll the next window of text onto the screen.

.= — Display the current line number.

Window Sizing The **ex** editor displays a window of text if requested. The z command is used to manipulate this window.

line z *type count* — Changes current line to *line* and displays a window consisting of *count* lines. The *type* specifies where the current line is displayed in the window. If no *type* is specified, then the line is placed at the top of the window. If *type* is a . then the line is placed in the middle and if it is a - the line is placed at the bottom of the window.

Examples:

:z	Redraw, place the current line at the top of the window.
:z .	Redraw, place the current line at the middle of the window.
:z -	Redraw, place the current line at the bottom of the window.
:/*pat*/z.	Redraw, place the line with *pat* at the middle of the window.
:z n	Redraw, use *n* line window.

Inserting Text The **ex** editor inserts text one line at a time. You can enter insert mode and enter text until you type a . and press Return at the beginning of a line. If you have the wrapmargin option set, you do not have to press Return at the end of each physical line when entering text.

line i — Insert text at line *line*. The insert command places text before the line of text currently at address *a*.

Example:

```
:5i
This is a new line of text.
.
```

Your new line of text becomes line 5 and the existing line 5 becomes line 6.

line a — Append text at line *line*. The append command places the text after the line of text currently at address *line*.

Example:

```
:5a
This is a new line of text.
.
```

Your new line of text becomes line 6 and the existing line 6 becomes line 7.

Changing Text The following command is used to change existing lines of text.

range c *count* — Enters insert mode and replaces the lines specified by *range* or *count* with the new text you typed. If you do not type any new text, the old text is deleted.

Example:

```
:7,9 c
These are new lines of text.
Inserted as line 7 thru 10.
Because we added an extra line.
```

This replaces lines 7,9 with the three new lines of text.

Copying Text The following command is used to copy existing lines of text to a new location or duplicate them.

range co *line* — Copies the lines specified by *range* to the line after *line*. A *line* of 0 (zero) copies text to the beginning of the buffer.

Example:

```
:5,21 co 73
```

Copies lines 5 through 21 and inserts them after line 73.

Deleting Text The following command is used to delete lines of text.

range d *reg count*

Delete the lines specified by *range* or *count* to the named register *reg*. The originals lines are deleted. Valid register names are a through z. If no register name is provided, the text is placed in the *undo* register. To retrieve text from registers use the *put* command. If you use capital letters for register names, the text is appended to the register.

Example:

```
:11,19 d a
```

deletes lines 11 through 19 to register a.

```
:32 d 8
```

deletes lines 32 through 40 (deletes the 8 lines following line 32).

Marking Positions The mark command allows you to create specific addresses. Thus you can create blocks of text based on these addresses. The one drawback to marks is that they are invisible. You mark a position in the buffer but you cannot see the mark.

line ma *x* Mark line number *line* as address or position *x*.

Example:

```
:ma a
```

marks the current line as position *a*.

'*x* Reference address or position *a*. Once you have a position marked you can use the position as an address for other commands.

Example:

```
:'a,'a+5 co 1
```

copies the lines from position *a* through position *a*+5 to line 1.

Moving Text The following command is used to move existing lines of text to a new location.

range m *line* Moves the lines specified by *range* to the line after *line*. A *line* of 0 (zero) moves text to the beginning of the buffer.

Example:

```
:5,21 m 73
```

moves lines 5 through 21 to after line 73.

Moving Text (cut & paste) By using the *yank* command you can yank any amount of text based on a range of lines. The yank command is referred to as cutting the text in some editors. The *put* command is used to put the text back in the buffer. You can also use the *move* command to move a range of text to a new location, see the previous command.

Yanking (cutting) The text may be yanked to the undo register or to a named register.

range ya *reg count* — *Y*ank the lines specified by *range* or *count* to the named register *reg*. The originals lines are not changed. Valid register names are a through z. If no register name is provided, the text is placed in the *undo* register. To retrieve text from registers use the *put* command. If you use capital letters for register names, the text is appended to the register.

range d *reg count* — *D*elete the lines specified by *range* or *count* to the named register *reg*. The originals lines are deleted. Valid register names are a through z. If no register name is provided, the text is placed in the *undo* register. To retrieve text from registers use the *put* command. If you use capital letters for register names, the text is appended to the register.

Example:

```
:ya 5
```

Yank five lines to the undo register.

```
:.,$ ya a
```

Yank all lines from the current line to the end of the buffer into register a.

Putting (pasting) The put command is used to retrieve text from a register. The text is not deleted from the register, thus is can be retrieved multiple times.

line pu *reg* — *P*ut data from register *reg* after line *line*. If *reg* is not specified then the undo register is used. If *line* is not specified then put text after current line.

Moving text between files You can move text from one file to another without leaving **ex**. You must first save the text to a named register, then switch to the other file, move to the location you wish to place the saved text, and then put it in the buffer.

range ya *reg* — Save the text to a named register (a-z).

e# — Switch to *alternate* buffer/file in the session buffer.

or

e *file* — Switch to a new *file* in the session buffer.

adr — Move to the desired location in the *alternate* or new file.

pu *reg* — *P*ut lines from register *a* after current line.

Indenting Text The following two commands are used to indent and deindent text defined by a certain motion.

range > *count* — Indent the lines specified by *range* or *count* by the number of spaces specified in the shiftwidth option.

range < *count* — Deindent the lines specified by *range* or *count* by the number of spaces specified in the shiftwidth option.

Filtering Text Filter commands take the text specified by the motion and send it to a UNIX filter. The text is deleted from the buffer and replaced by the output of the UNIX filter.

range!*cmd* — Send *range* lines of text to standard input of any UNIX filter command to be manipulated and then returned; the original text is replaced. A filter command must be standard input and output capable (e.g., grep, sed, sort, etc.)

An example is well worth explaining for this command. Assume you have 20 lines of text you need to convert into two-column format instead of the one-column format you currently have in the buffer. To do this you just type in the following command

```
.,+19!pr -t2
```

and press Return. The .,+19 range takes all of the text from your current line plus the next 19 lines and sends it to the **pr -t2** command. The **pr -t2** command reformats the input text into two-column output. Your screen is redrawn with the same data in two-column format. If you don't like the way it looks you can always press **u** for undo.

Join Lines The following command is used to join lines together. It removes the new-line that forms a physical line.

range j *count flags* — Joins the lines specified by *range* or *count* to create one line. If no *range* or *count* are given, the current and next line are joined. If a line ends with a sentence punctuation (., !, ?) two spaces are placed between the last character of the line and the next line joined to it. If the next line begins with a right parenthesis ()) no spaces separate the joined lines.

range j! *count flags* — Same as the j command except no space adjustments are made to the lines that are joined.

Example:

```
:%p
This is line one.
This is line two.
:1 j p
This is line one.  This is line two.
```

Undo, Redo, and Retrieve The following commands are used to undo previous commands, repeat previous commands, or retrieve text that was deleted or saved earlier in the edit session.

u	Undo last change.
U	Undo all changes on current line. Must not move off of the current line.
.	Repeat last command.
&	Repeat last **ex** substitute command.
[1-9]pu	Retrieve 1-9th last delete from a number register.
pu *reg*	Retrieve previously stored contents from a letter register.
1pu.u.u	Scan the number registers for last deleted text.

Edit/Re-edit a File These commands are used to edit different files without leaving the editor. You can switch between various files by using the e command.

e +*cmd file* — Edit *file* after interpreting **ex** command *cmd*. If the current buffer has been modified since the last save to disk (:w), **ex** displays a warning and does not edit the new *file*. If *cmd* is not provided, the editor enters the new buffer at the last line of the file. The *cmd* must be an address command.

e! +*cmd file* — Edit *file* regardless of unsaved changes in current buffer. All modifications you have made since the last save to disk are lost. If no *file* is specified then the current file is reedited.

e# Edit alternate *file* if two files are being edited. This does not affect the argument list. If you are editing five files and you decide to edit a different file using the e command, you can use the e# command to return to the previously edited file. If you use the n command, the next file in the argument list is edited.

Examples:

`:e text.c`	Replace the current file being edited with the file *text.c*.
`:e +/main/ myprog.c`	Edit *myprog.c* and set the current line to the first line containing the word *main*.

Edit/Re-edit Multiple Files These commands allow you to edit a new set of files and move between the different files.

n *newargs* Replace current argument list with new list of file(s) specified by *newargs*. If no arguments are specified, then the next file in the argument list is edited. If the current buffer has been modified since the last save to disk, ex displays a warning and does not edit the next file.

n! Edit *n*ext file in argument list regardless of unsaved changes. All modifications you have made since the last save to disk are lost.

ar Display current *ar*gument list of files to be edited ([current file] is enclosed in brackets).

rew Rewind argument list to beginning, reedit first file.

rew! Rewind argument list to beginning, reedit first file regardless of unsaved changes.

Examples:

`:n myfile yourfile afile newfile`	The argument list maintained by ex is reset to the four files listed. The file *myfile* is placed in the buffer for editing.
`:n`	The next file in the list is edited. In this case *yourfile* is placed in the editor's buffer.
`:ar`	Display the current list of files being edited. The file *yourfile* is enclosed in brackets since it is the file currently being edited.
`:rew`	Rewind the argument list to the beginning and edit the first file listed.

Read from a File The r command allows you to read from a file or the output of a UNIX command.

line r *file* [...] Read in file(s) to current buffer after line *line*. If *line*=0 then file is placed at the beginning of buffer.

line r !*cmd* Read in output of UNIX command(s) and place after line specified by *line*.

Examples:

`:0r !ls -x`	Reads the output of the ls - x command and places it at the beginning of the buffer.
`:$r ../C/myprog.c`	Reads the file ../C/myprog.c in at the end of the buffer.

Write to a File The w command allows you to write specified lines of the buffer to a file or to the input of a UNIX command.

range w *file* Write lines specified by *range* to the file named *file*. If *range* is omitted, then the entire buffer is written. If *file* is omitted, the current filename is used. If no current filename is known, then *ex* complains, not knowing where to write the contents.

range w! *file* Write lines specified by *range* to disk, overwriting the existing file.

range w >> *file* Append lines specified by *range* to disk *file*.

range w !*cmd* Write the *range* of lines to the UNIX command. The command will process the input.

Examples:

`:w!`	Write current buffer to disk, overwriting existing file.
`:w #`	Write to the alternate file. You must have an alternate file active; using the :e, :n, or :w command will set the last filename used to the alternate file.
`:w >> TMP.%`	Append current buffer to disk file TMP.*currentfile*.
`:w %.old`	Write current buffer to file *currentfile*.old.

Escape to Unix These commands allow you to interface with the UNIX utilities. You can temporarily exit to a shell without leaving the editor among other various functions.

sh Escape to a subshell, return to **vi** with a Ctrl-D, or exit.

!*cmd* [*args*] Escape to shell and perform single UNIX command with arguments.

range w !*cmd* Escape to shell and perform single UNIX command on lines specified by *range*.

!! Repeat last ! command.

range !*cmd* Send lines specified by *range* to UNIX command and replace specified lines with output. See Filtering Text for further discussion.

stop Stop editor, place in background and return to top level shell. Berkeley UNIX only, **fg** command resumes editing.

Ctrl-Z Stop editor; same as the **stop** command.

!cmd % Escape to shell and perform single UNIX command with current file as an argument.

Searches These commands search for the first occurrence of a pattern.

NOTE

Refer to Module 113 on Regular Expressions for special characters used for *pat*terns in the search and substitute commands. For a brief synopsis of the regular expressions supported by **ex** and **vi** refer to the following section on Regular Expressions.

NOTE

The slashes (/) used in the following search and substitute commands are delimiters for the search, old, and new strings. You can use any character as the delimiters for these commands.

:/pat/cmd Perform **ex** command *cmd* on line matching first occurrence of *pat*.

Examples:

`:/John`	Search for first occurrence of John. No command is given so nothing is done; only the current line is changed to the new location.
`:?Tina`	Search backward for first occurrence of Tina.
`:/John/p`	Search for first occurrence of John and print the line.
`://`	Repeat last search; search for John.
`:??`	Repeat last search backward.

Global Searches The following commands search the entire file or range of lines specified and match each line containing the given *pat*tern.

NOTE

A search command (global) may be continued on the following line by typing a backslash (\) followed by a Return.

```
:g/Test 1/s//Test 2/p\
i\
# RJF\
.
```

Searches for all lines containing the pattern Test 1 and substitutes Test 2 in place of Test 1. Inserts the # RJF line before each line that was substituted.

range g/pat/cmd Perform **ex** command *cmd* on each line specified in *range* that has a matching *pat*. If *range* is not defined, then all lines are searched for a match. If *cmd* is not given, then the current line becomes the last line containing the string *pat*.

Examples:

`:g/Total/`	Search for last occurrence of Total. The line is not displayed on your screen.

`:g/pat/nu` Print all (global) occurrences of *pat* preceded with the line numbers.

`:g/ME/s//I/gp` Searches for all occurrences of ME and changes them to I and prints the line. The g in gp specifies global line substitution, which means all occurrences on a line are changed.

Global Exception Search The following commands search the entire file or range of lines specified and match all lines NOT containing the given *pat*tern.

range v/*pat*/*cmd* Perform ex command *cmd* on each line specified in *range* that does NOT have a matching *pat*. If *range* is not defined, then all lines are searched for a NON-match. If *cmd* is not given, then the current line becomes the last line containing the string *pat*.

Examples:

`:v/Total/` Search for last line that does not contain an occurrence of Total. The line is not displayed on your screen.

`:v/Tom/nu` Print all (global) lines that do not contain an occurrence of Tom. Each line is precede by its line numbers.

`:v/ME/s/You/I/nu` Searches for all lines that do not contain the ME pattern and changes the You pattern to I and prints the changed line preceded by a line number.

Substitutions The substitute command is useful for changing specific portions of a line for a specified range of lines.

range s/*old*/*new*/*opt count flags*
Substitute *old* pattern with *new* on lines specified by *range* or *count*. If the range or count is not specified, the current line is assumed. The options may be c or g. See Substitute options that follow. The *flags* may be #, nu, p, l, +n, or -n. Refer to previous Command Format section.

range & *options count flags*
The & replaces the last s/old/new/ substitute command. You can also use the *s* in place of the &.

Substitute Options These options may be placed at the end of a substitute command.

c Interactive confirmation (answer y or n).

g Global substitutions for entire line.

Examples:

`:s/We/Us/g` Substitute all occurrences of *We* with *Us* on the current line.

`:/Test/s/1.0/1.1/p` Search for first line containing *Test*, then substitute first occurrence of *1.0* with *1.1* and print the changed line.

`:100,$s/outlaw/inlaw/c`	Search lines 100 through end-of-file for *outlaw* and prompt to change the first occurrence to *inlaw*. If you respond to the prompt with a y then the substitution is done.
`:~`	Repeat the last substitution.

REGULAR EXPRESSIONS The following table contains each regular expression and the task that it performs when used inside a pattern. A regular expression consists of regular alphanumeric characters matching themselves and special characters matching certain patterns of text. Regular Expressions are often referred to as REs in UNIX terminology. Thus we use the RE notation for uniformity and briefness. For further descriptions refer to Module 113, Regular Expressions.

Alphanumeric RE	*Description*
c	Matches any ordinary character *c*.
string	Matches the set of characters *string*.

Metacharacters Metacharacters are the special characters used in regular expression patterns that have special meanings. Metacharacters are often referred to as *special* or *magic* characters or wild cards.

Special RE	*Description*
\	Escapes the meaning of a metacharacter.
^	Matches the beginning of the line.
$	Matches the end of the line.
.	Matches any single character.
[*class*]	A character class. Matches any one character in the class.
[*c1-c2*]	Match any one of the ASCII characters in the range defined within the brackets.
[^*class*]	Do NOT match any of the ASCII characters listed within the brackets. Ranges may be specified.
*	Matches zero or more occurrences of the preceding regular expression.
\<	Matches the beginning of a word.
\>	Matches the end of a word.
\(\)	Groups the regular expression RE for later reference; each parenthetical RE is referenced by a corresponding number (see \n). The first group is referenced by \1, the second group by \2, and so on. You can have a maximum of nine groups in one regular expression pattern.
\n	Matches the *n*th grouped \(RE\) withing the same regular expression.
//	Uses the last RE if it has been defined.

To have a metacharacter interpreted as a normal character precede it with a backslash (\).

Special Substitute Characters There are special characters that you can use in the replacement (new) string of a substitute command to help speed up the process.

&	Matches last search or substitution (old) string.

Example:

```
:s/born/new&/p
The newborn is doing fine.
:s/Tom/&my/p
My name is Tommy.
```

The first command substitutes *born* to *newborn*. The second substitutes *Tom* to *Tommy*.

~ Matches last replacement (new) string.

Example:

```
:s/finish/end/
:s/stop/~/
```

The first line substitutes *end* in place of *finish*. The second command substitutes the last replacement string, *end* in place of *stop*.

\l Converts the next character in the replacement string to lowercase.

Example:

```
:s/A big/\l&/p
a big
```

The substitute command replaces the string *A big* with the string *a big*.

\L Converts all following characters of the replacement string to lowercase until a \e or \E is encountered.

Example:

```
:s/THIS/\L&\E/p
this
```

\u Converts the next character in the replacement string to uppercase.

\U Converts all following characters of the replacement string to uppercase until a \e or \E is encountered.

n The *n*th grouped \(RE\).

\E or \e Turn off the action of the \U or \L strings.

TAGS Tags provide the ability to edit multiple files via symbolic labels. These symbolic labels or tags are simple strings that you select. There are several steps to making the tag command work.

1. You must set up a tag file, which consists of lines with three fields.

Field	*Description*
1	The tag (symbolic label)
2	The file to edit when that tag is specified
3	The ex commands to interpret when the file is edited

2. You must set the tag reference option in one of the following ways.

   From within ex/vi,

```
:set tags=/ul/tech/mylogin/.tags
```

In your .exrc file,

```
:set tags=/u1/tech/mylogin/.tags
```

In your EXINIT variable,

```
EXINIT="tags=/u1/tech/mylogin/.tags"
```

3. You request a tag with the tag command,

```
:tag prog1
```

Tag File Format The format of the tag file is important; it must contain tag names (symbolic names) sorted in alphabetical order. The second column is the name of the file to be edited. The third column contains the **ex** edit commands to be processed when the new file is edited. For example, the following is the generic format of the tag file.

```
label<Tab>file<tab>ex_commands
```

An example of a tag file follows:

```
main    #       /main()/;.-5,.p
```

which says when you request tag main, the editor will edit the alternate file, search for main(), and display the 5 preceding lines and the current line.

See **ctag(1)** to build a tag file from C source code files.

Referencing a Tag To reference a tag in the tag file you enter **tag** *tag*, where *tag* is the tag you wish to reference, and press Return. For example,

```
:tag label
```

This causes **ex/vi** to check the tag file for the tag *label* in field one. If the *label* tag is located then **ex/vi** edits the file specified in field two. Once the file has been read in by **ex/vi** the **ex** commands in field three are performed. Then you are placed at the location after the commands have been carried out.

The following example references the *main* tag defined early. Type

```
:ta main
```

and press Return. This tag command would cause **ex/vi** to edit the alternate file (#), search for the pattern *main( )*, and print out the 5 lines before *main()*.

NOTE

The tag file must be in alphabetical order for the tag command to function correctly. If it is not sorted, the tag command will complain about tags not being found.

MACROS Macros are a special feature of the **vi** editor that allows you to store multiple keystrokes and reference them with a single or shorter set of keystrokes. You can have multiple **vi** or **ex** commands defined in one macro. Macros are discussed in Module 151.

MISCELLANEOUS The following list is composed of commands that provide various functions.

range # *range* nu	Display the line *nu*mber and the line. If no *range* is provided the current line is used.
.=	Display the line number of the current line.

=	Display the total number of lines in the buffer.
ar	Display current file and files to be edited. The current file being edited is enclosed in brackets.
cd *dir*	Change current working directory of editor. The editor performs a UNIX cd command internally.
pre	Preserve the file for later recovery using the recover command.
rec	Recover the file from a previous preserve. You can preserve the file if you like, but usually preserving is done automatically when your phone line is disconnected, your terminal hangs, or the system crashes.
so *file*	Read and execute commands in *file* to set/reset editor options. The commands in the file must be **ex** type commands. Refer to the section that discusses the .exrc file.
stop	Stop editor, place it in background, and return to top level shell. Job Control UNIX only, use the **fg** command to resume editing.
ver	Displays the version of **ex/vi** editor you are using. Believe it or not, different versions have minor differences in their support of features.
cmd1 | *cmd2*	Sequential editor command separator. Perform editor command *cmd1* followed by editor command *cmd2*. This is not a UNIX pipe command.
cmd1 ; *cmd2*	Same as above, but only usable in tag files.

LEAVING ex There are multiple ways to leave the **ex** or **vi** editor. All of the following commands can be used from **ex** or **vi**.

ZZ	Save the current session buffer to disk if any modifications have been made and exit the editor.
x *file* wq *file*	Save the current session buffer to disk and quit **ex/vi**. If *file* is specified, the buffer is written to that file instead of the current buffer name.
q	Quit **ex/vi** if no modifications have been made since the last save to disk.
q!	Quit **ex/vi** without checking for unsaved modifications. If you have made any changes to the session buffer, they are discarded without warning. This may be useful if you have destroyed vital information.
sh	Leave **ex/vi**; starts a subshell. Press **Ctrl-D** or type **exit** to return to **ex/vi**.
Ctrl-Z	Stop the editor, place it in background, and return to top level shell. Job Controlled UNIX only, the **fg** command resumes editing.

CUSTOMIZING OPTIONS **ex** has internal variables (switches) which allow you to **set** options for special effects or customizing of **ex/vi**. There are three types of these variables:

- Boolean, toggle on or off (:set option)
- Numeric, assign a numeric value (:set option=val)
- Alphanumeric string, assign a string value (:set option=str)

There are three ways to *set* variables:

- Setting the '*EXINIT*' shell variable
- Defining the values in the *.exrc* file
- Using *:set* while in *ex/vi*

The following commands are used with **ex/vi** to display the options.

set	Display only options that have been changed.
set all	Display all options and current setting.
set [opt] ?	Display current setting for given [opt].

The Options Precede each of the following with *set*. The abbreviated version of each variable follows in ( ) if it exists.

Option	*Default*	*Description*
autoindent(ai)	noai	**ex/vi** will indent the next line the same as current line while in insert mode. Ctrl-T moves the indention right one tab or Ctrl-D moves it left one tab.
autoprint(ap)	ap	Displays the current line after each **ex** command that modifies text. The lines affected by the g and v options are not displayed.
autowrite(aw)	noaw	Causes an automatic write to disk if :!, :tag, :next, :rewind, Ctrl-^, Ctrl-[, or Esc commands are used with unsaved changes.
beautify(bf)	nobf	Discards all control characters except tab, new-line, and form-feed from the input.
directory(dir)	dir=/tmp	Directory where **ex/vi** stores temporary buffer and register files. The editor must have read and write permissions for the directory.
edcompatible(ed)	noed	Causes **ex** to change how it handles the substitute suffixes g and c. The g and c options used on substitute commands become toggles. The first use of g causes all subsequent substitutes to be global in effect until the next g is used. The c is for confirmation of substitutes.
errorbells(eb)	noeb	Precede error messages by a bell. For the sleepy user or the visually impaired.
hardtabs(ht)	ht=8	Specify hardware tab spacing.
ignorecase(ic)	noic	Map all uppercase letters to lowercase letters in regular expression matching. In other words, ignore whether a letter is uppercase or lowercase when performing searches.

Option	*Default*	*Description*
lisp	nolisp	Autoindent for LISP code. The |, [[, and]] are changed to affect s- expressions and functions.
list	nolist	Display tab and new-line visually. Tabs appear as ^I and the end-of-line (new-line) appears as $.
magic	magic	Enable metacharacters for matching. If magic is set then ., *, [, and \ are considered to be *magic* (metacharacters) in patterns used for searches and substitutions.
mesg	mesg	Allow messages from other users' commands, such as *write* and *talk*, to interrupt your edit session. For those who like to communicate with others. See the UNIX *mesg* command.
novice		Set by invoking the editor as **vedit** for beginner mode. Displays when you are in insert mode and various other help type messages.
number(nu)	nonu	Line numbering. Line numbers are displayed to the left of each line in the buffer. The line numbers are only on the display; they are not placed in the text.
open	open	Allow open mode from **ex** mode. Allows you to enter open window mode from the **ex** editor mode.
optimize(opt)	opt	Tries to speed up **ex** formatting if possible.
paragraphs(para)	para=IPLPPPQPP LIpplpipbp	Set paragraph delimiters used by the { and } commands. The IPLPPPQPP LIpplpipbp string considers the .IP, .LP, .PP, .QP, .P , .LI, .pp, .lp, .ip, and .bp to be paragraph delimiters. If you use a space as a delimiter, precede it with a backslash (\).
prompt	prompt	The : prompt will be printed when in **ex** mode.
readonly(ro)	noro	Set to read-only mode, disables write capability. The **view** command invokes **vi** with this option set.
redraw(re)	nore	Keep dumb terminals updated continuously. Normally **vi** inserts an @ sign in place of a deleted line on a dumb terminal. If Redraw is set, the screen is updated to remove these deleted lines. Extremely costly on output, but if you have a dumb terminal and a high baud rate it's nice.

Option	*Default*	*Description*
remap	remap	Allow macros within macros. If a is mapped to b, and b is mapped to c, then a is mapped to c automatically.
report	report=10	Report modifications at the bottom of the screen if the set number of lines are modified.
scroll	scroll=11	Number of lines to scroll when using Ctrl-U or Ctrl-D commands.
sections(sect)	sec=NHSHHHU	Set sections delimiters used by the [[and]] commands. Works the same as the paragraphs option.
shell	shell=/bin/sh	Name of shell to use when a shell is invoked from **vi**. The $SHELL variable is used if it is defined and exported.
shiftwidth(sw)	sw=8	The number of spaces the text is indented by using the Ctrl-T, Ctrl-D, and Tab keys and the < and > commands and the *autoindent* option.
showmatch(sm)	nosm	In the *vi*sual mode this option causes the editor to show the matching (or { for every } or) you typed.
showmode	noshowmode	Shows the current mode of the editor; informs you if you are in INSERT, COMMAND, or EX mode. Some editors do not support this option.
slowopen(slow)	slow	Slows the updating of dumb terminals. During insertions the text is not updated to resemble the immediate changes. Speeds up overall performance at the expense of clarity. Just remember to use Ctrl-R or Ctrl-L enough to know what you are changing.
tabstop(ts)	ts=8	Expand tab to boundaries that are multiples of the value provided.
taglength(tl)	tl=0	The number of significant characters in a tag; 0 means all characters are significant.
tags	tags=/usr/lib/tags	Set directory path of tag file.
term	$TERM	Terminal type used by **vi** for screen control. Cannot be set in **vi**. May enter a Q, set term, then reenter **vi**.
terse	noterse	Short error diagnostic messages. As if they were ever explanatory enough.
timeout(to)	noto	Allows a shell or system timeout to abort the edit session.

Option	*Default*	*Description*
ttytype(tty)		Terminal type. Left over from the dark ages when **vi** was still a dream.
warn	warn	Warns of escaping to the shell without a current save to disk of the buffer.
w*XXX*	*stty BAUD*	Set window size according to baud rate XXX. Valid wXXX are w300, w1200, w2400, w4800, w9600.
window	window=14	Set number of lines in window. Initial set of the window size uses the wXXX option. Each time you move from the current window and require a new window to be drawn, *n* number of lines are used for the initial window.
wrapmargin(wm)	wm=0	Automatic new line inserted *wm* spaces from the right margin during insert mode. It is advisable to place this option in your EXINIT variable. Long lines can cause problems in document processing.
wrapscan(ws)	ws	Searches will wrap around the end of the file and continue until the current line is reached again.
writeany(wa)	nowa	No safety checks are done before saving a file. The contents of the session buffer are written to disk without checking current status of disk file or buffer condition. The good side is you always have a copy on disk. The bad side is you may write something to disk you don't want.

The .exrc File The *set* options can be placed in the .exrc file so that each time *vi* is invoked the options are read and *set* automatically. The *.exrc* file may be placed in the user's HOME directory or the current working directory. The options *set* in *.exrc* will be overridden by the settings in the EXINIT variable if it exists.

An example *.exrc* file:

```
cj> cat .exrc
set ai | map v :w
set window=5
set wm=8
map Ctrl-Z :shCtrl-M
ab rf Robert Felps
```

These commands activate auto-indent (in the .exrc file | is equal to a new-line), map v to :w. Set the window size to 5 lines. The report line will be updated if 1 or more lines are modified. Set the right margin to 72; new-lines are inserted automatically. Map control-Z to an escape to a subshell key sequence. You must use Ctrl-V then Ctrl-M when creating the map. Abbreviate rf for Robert Felps.

You may save *ab*reviations, *maps*, and *set* commands in any file. For **ex/vi** to read them during initialization the file must be named .exrc and reside in the current working directory. If the commands reside in another file, you can have **ex/vi** read the file at any time by using the following command.

```
so file
```

where file is the name of the file containing the commands.

The EXINIT Variable The shell variable (EXINIT) can be assigned **vi** options. When **vi** is invoked the options are read and *set* inside **vi**. The EXINIT variable should be put in your .profile file so it is *set* when you log in. The EXINIT variable will override the *.exrc* file settings.

The EXINIT variable can be *set* as follows:

```
cj> EXINIT='set ai window=5 report=1 wm=8'
```

This command sets auto-indent, the window size to 5 lines, causes updates to the report line if 1 or more lines are modified, and sets the page width to 72 columns with auto-wrap margin.

DIAGNOSTICS AND BUGS There are a few bugs or implementations of features that are worth noting.

The undo command (u) causes all marks to be lost on lines that were involved in the change.

The undo command never clears the modified buffer status.

No warning is given of text stored in named registers when you exit the editor.

The z command prints a logical number of lines, not the physical. Thus if you have long lines that wrap around the screen, the top lines of the screen may scroll off the visual terminal screen.

The NUL (octal 000) is discarded on input and thus cannot be entered in the buffer.

The new-line (012), Ctrl-J, character cannot be placed in the text as visual characters.

RELATED COMMANDS Refer to the **nawk**, **ed**, **sed**, and **vi** commands described in modules 6, 39, 117, and 151.

RELATED FILES The following list of files are used by **ex/vi**.

/usr/lib/ex?.?strings	The error messages used by **ex/vi**
/usr/lib/ex?.?recover	The recover command
/usr/lib/ex?.?preserve	The preserve command
/usr/share/lib/terminfo/?/*	Terminfo database for terminal descriptions
$HOME/.exrc	The startup file used to store the **ex/vi** environment
./.exrc	The startup file used to store the **ex/vi** environment
/tmp/EX*nnnnn*	Temporary buffer copy of the file
/tmp/Rx*nnnnn*	Temporary named register storage files
/var/preserve/*login*	Directory where preserved files are kept

APPLICATIONS

The **ex** command is useful in performing editing functions that do not require full-screen viewing of the data. For example, global substitutions, file manipulation within the editor, macros, abbreviations, tags, and various other features are beneficial to the everyday requirements of editing.

TYPICAL OPERATION

In this activity you use the **ex** command to create and edit multiple files. Begin at the shell prompt.

1. Type **ex mypasswd mygroup** and press **Return**. This starts **ex** with a buffer named *mypasswd*. Since no file by that name exists, the buffer is empty.

   ```
   cj> ex mypasswd mygroup
   ```

2. Now read in the /etc/passwd file by typing **0r /etc/passwd** and pressing **Return**.

   ```
   :0r /etc/passwd
   ```

3. To view the entire contents of the file type **%p** and press **Return**. Use Ctrl-S and Ctrl-Q to start and stop the display as desired.

4. Create a backup copy of the buffer by writing the contents to a file on disk. Type **w %.bak** and press **Return**.

   ```
   :w %.bak
   ```

5. Type **/mylogin** and press **Return** to locate the line containing your login. Replace mylogin with your real login name.

6. Make a copy of this line by typing **ya** to yank the line to the undo register. Then type **pu** to put the line back in the session buffer.

7. Display the new line by typing **p** and pressing **Return**.

8. Change all the colons on the current line to tabs. Type **s/:/ /g** and press **Return**. The line should be displayed after the substitution is performed.

   ```
   :s/:/    /g
   ```

9. Type **u** and press **Return** to undo the substitute command.

10. Insert a line at the beginning of the buffer by typing **1i** and pressing **Return**.

11. Insert the following line of text. To exit insert mode type a **.** in the first position of a line and press **Return**.

    ```
    NEWUSER::5000:50:New User:/u1/new/user:/bin/ksh
    .
    :
    ```

12. Save the file to disk by typing **:w** and pressing **Return**.

13. Save the first 5 lines of the buffer to register a. Type **"1,5 ya a** and press **Return**.

14. List all the files being editing by typing **arg** and pressing **Return**.

    ```
    :arg
    [mypasswd] mygroup
    ```

15. Type **n!** to edit the next file *mygroup* without having **ex** check for unsaved modifications of the current file.
16. Type **0r /etc/group** and press **Return** to read the */etc/group* file into the buffer.
17. Type **%s/:/ /g** and press **Return** to substitute all : (colons) to tabs.
18. Type **%nu** and press **Return** to display the contents of the buffer with preceding line numbers.
19. Type **pu a** and press **Return** to put the 5 lines you saved from the passwd file into buffer a back into the current buffer.
20. Type **%nu** and press **Return** to display the contents of the buffer again. Notice the 5 new lines at the bottom of the buffer.
21. Type **q!** and press **Return** to quit the editor without saving the changes.
22. Type **rm mypasswd*** and press **Return** to remove the files you just created.
23. Turn to Module 151 to continue the learning sequence.

Module 44
exec

?PC

DESCRIPTION

The internal **exec** command replaces the current shell process with the specified command. Normally, when you run a command a new process is spawned. The **exec** command does not spawn a new process. Instead, the current process is overlaid with the new command.

It may also be used to set redirection file descriptors. In this situation the current shell process is modified but not overlaid.

COMMAND FORMAT Following is the general format of the **exec** command.

```
exec [ command ] [ arg ... ]
exec fd< file
exec fd> file
```

Arguments The following list describes the arguments that may be passed to the **exec** command.

command	An executable command. The command replaces the current shell in memory without spawning a new process. If you cannot execute the command for some reason, then you exit the current shell.
arg	Arguments to command.
fd< file	Opens *file* for input with file descriptor *fd*.
fd> file	Opens *file* for output with file descriptor *fd*.

C Shell

The C shell does not support using exec to change file descriptors (fd).

FURTHER DISCUSSION If you specify a command, the shell replaces itself in memory with the new command you specified. For example,

```
exec vi myfile
```

replaces the shell with the **vi** program text and places you in vi, editing myfile. When you exit **vi** you exit the system because **vi** has become your login interface to the operating system.

Redirection You can use **exec** to perform I/O redirection commands. The following examples illustrate the use of **exec** for redirection purposes.

exec 3< inputfile	# Opens inputfile with file descriptor 3 for reading.
exec 4> outputfile	# Opens outputfile with file descriptor 4 for writing.
exec 5<&0	# Makes fd 5 a copy of fd 0 (standard input).
exec 6>&p	# Attach fd 6 to co-process.

RELATED FILES The **exec** command can be used to open, close, and copy file descriptors. If you do not specify a command or arguments, you can specify redirection symbols and file descriptors to perform these functions.

APPLICATIONS

You can use the **exec** command to replace the current shell without creating a new process. This becomes useful in a set of shell scripts that execute one another. Instead of spawning new processes each time you call a different shell, you can **exec** to the new shell.

The **exec** command may also be used to open, close, and copy file descriptors as specified by I/O redirection.

TYPICAL OPERATION

In this activity you create a script which execs mailx or an editor. Thus when you exit mailx or the editor you return to your shell not this script.

1. Type the following lines using the editor of your choice.

```
while true
do
echo "Go to Mail, Editor, or eXit:"
read ANS
case "$ANS" in
        [mM]) exec MAILX ;;
        [eE]) exec ${EDITOR:=vi} ;;
        [xX]) ezec exit ;;
esac
done
```

2. Change the modes of the script by typing **chmod 755 ash** and pressing **Return**.
3. Now run the script. Type **ash** and press **Return**.
4. Press **Return**. Notice the prompt reappears. Now type **e** to enter the editor. Now exit your editor. Notice the prompt did not reappear. You exited back to the main shell not the script because the script was overlayed by the editor.

C Shell

The **csh** exec command works the same way. You could write the same script just by changing the syntax for csh.

---

5. Turn to Module 48 to continue the learning sequence.

Module 45
exit (csh, ksh, sh) / logout (csh)

DESCRIPTION

The internal **exit** command causes the current shell to terminate. If the current shell is a child process, execution returns to the parent process when it exits. If the current shell is the login shell, you're logged off the system when the shell terminates.

The **Ctrl-D** character can be used to perform the same function. The Ctrl-D informs the shell it has reached the end-of-file (EOF). Because the shell is a filter and filters terminate when EOF is reached, the shell terminates and you log off the system or return to the parent process.

If you are writing a shell script and you need to provide a return code (exit status) to a parent process, you can use the **exit** *n* format, where *n* is the return code you want returned to the calling (parent) program.

COMMAND FORMAT Following is the general format of the **exit** and **logout** commands.

```
exit [ n ]
Ctrl-D
```

C Shell

```
exit [n]
logout
Ctrl-D
```

Options Only one option may be used to control how **exit** functions.

n The return code (exit status) you wish to return from the termination of the current shell

FURTHER DISCUSSION To enter a subshell you have to run the **ksh**, **csh**, or **sh** program. For example, you would type

```
cj> ksh
```

and press Return. To return to your parent shell you type

```
cj> exit
```

and press Return. The **exit** and Ctrl-D are handled a little differently by the shell. An **exit** command is processed as a normal internal command. Whereas the Ctrl-D is interpreted as EOF (end-of-file) and causes the shell to exit immediately. By using **exit** you can specify a return code value. This value is stored in the parent shell variable $? ($status-csh). An example of how you might use this return code and variable will help explain return codes.

Suppose you want to write a script named *readable* that returns a 0 if a file is readable and a 1 if the file is not readable.

```
if [ -r "$1" ]
then exit 0
else exit 1
fi
```

You pass *readable* a single argument, which is a filename. If the file is a readable file, the return code is a 0; otherwise, a 1 is returned. Now in your parent shell you can use the $? ($status-csh) variable to test whether the file is readable are not. For example, your parent shell might look like this:

```
readable somefile
if [ $? ]
then echo "Processing the file"
...
else echo "Cannot read the file"
fi
```

This provides you with the capability to tell whether a child shell script completed with or without errors.

C Shell

The **logout** command is an internal **csh** command that logs you off the system. It only works while in the login shell. If you are in a subshell, the logout command will not perform an exit or logout.

When you type **logout** and press **Return** in your login shell, the shell executes the commands in the ~/.logout file before exiting.

The **logout** command can be duplicated in the **ksh** by setting an alias that performs an **exit** for you. Some systems implement **logout** in the shell, whether it be **sh** or **ksh**. It is not implemented on all UNIX systems; therefore, you may want to create it as one of your own aliases. You can make the **logout** alias perform other functions before logging you out of a shell or off of the system. Refer to Module 4 for more details on **alias**. The following example is a possible **logout** alias:

```
cj> alias logout="clear; du -s $HOME; times; exit"
cj> logout
```

In this example your screen is cleared, your disk space usage is displayed, and the shell status information is displayed before you are logged off the system.

Another alternative is to set a trap in your ~/.profile to execute the ~/.logout file. Simply place the following line in your ~/.profile.

```
trap '. $HOME/.logout; exit 99' 0
```

When you log out of the system with the **exit** command, the shell will trap signal 0 and execute the $HOME/.logout file and then exit the shell.

Logging out of the system is important. By logging out you prevent unfriendly people from causing possible mischief, using your terminal and user ID. If you are away from your terminal for more than a few (5) minutes, log off! If your terminal resides in a secure area, this may not be that important, but it's better to form the habit early than be sorry later.

Logging out provides someone else with a free port (/dev/ttyXX) to use if you are dialing in on a modem. If you are hardwired, this may not make any difference. Some UNIX systems are configured to log you out if you turn your terminal off. Terminals connected via modems usually perform in this manner. Check with your system administrator before you turn your terminal off to log out when using a modem. If the system does not recognize that you dropped the carrier on the modem connect, you will not be logged out. Therefore, the next person that dials in on that system modem will receive your shell prompt instead of the system's login prompt.

RELATED COMMANDS Refer to Module 71 describing **ksh**.

RELATED FILES The **exit** and **logout** commands do not display output unless you have them aliased to perform some pre-exit commands.

APPLICATIONS

The **exit** and **logout** commands and the **Ctrl-D** character provide a means of exiting or logging off of the system when you are ready to stop using it. Any of these three can be used to exit subshells and return to the previous shell and environment. The **exit** command is also used to exit a shell script without completing the following commands.

TYPICAL OPERATION

In this activity you use the **ksh** and **exit** command to create a subshell and then exit back to your login shell. Begin at the shell prompt.

1. Type **ksh** and press **Return**. Notice a new prompt is displayed on your terminal screen. You are now in a subshell.

C Shell

Type **csh** and press **Return**.

---

2. To return to your login shell, type **exit** and press **Return**. You will see your login shell prompt appear.
3. Turn to Module 92 (SV), Module 100 (BSD) to continue the learning sequence.

Module 46
expand/unexpand (BSD)

DESCRIPTION

The external **expand** command expands tabs to spaces. The **expand** command reads the named files or the standard input, changes the tabs to the appropriate number of blanks, and writes the results on the standard output.

The external **unexpand** command replaces leading spaces and tabs with just tabs where possible.

COMMAND FORMAT Following is the general format of the **expand** command.

```
expand [ -tabstop | -tab1,tab2,tab3,...,tabn ] [ file_list ]
unexpand [ -a ] [ file_list ]
```

Options The following options may be used to control how **expand** functions.

-tabstop	Sets tab stops *tabstop* spaces apart.
-tab1,.,tabn	Sets tabs at specific tab positions defined by *tab1* through *tabn*.
-a	Tabs are inserted at all occurrences that would replace two or more spaces. Without the -a option unexpand only replaces leading blanks and tabs.

Arguments The following argument may be passed to the **expand** command.

file_list	The list of files read and processed by **expand**. If no files are specified **expand** reads the standard input.

RELATED COMMANDS Refer to the **tabs** command described in Module 130.

RELATED FILES The **expand** command reads from standard input or files and writes to standard output.

APPLICATIONS

You use the **expand** command to convert tab characters to the appropriate number of spaces. You may find **expand** useful to preprocess character files that contain tabs before performing character processing on the files. For example, you may want to remove tabs before sorting or examining specific columns of a file.

TYPICAL OPERATION

In this activity you use the **expand** command to remove tabs from a file containing tab-separated columns. Begin at the shell prompt.

1. Type **cat -v -t file1** and press **Return**. Notice the ^I (tabs) embedded in the output.
2. Type **expand file1 | cat -v -t** and press **Return**. Notice the output has no tabs in it.
3. Turn to Module 143 to continue the learning sequence.

Module 47
export (sh, ksh)

DESCRIPTION

The internal **export** command places variables in the shell environment. The shell maintains two sets of variables, the local variables and the global or environment variables. Local variables are only known to the current shell. Environment variables are know by the current shell and are available to all processes started by the current shell. The **export** command marks local variables to become environment variables.

COMMAND FORMAT Following is the general format of the **export** command.

```
export [ variable ... ]
```

Arguments The arguments for the **export** command are variables that you wish to place in the shell environment. You can export a variable without setting it to a value. When it does become set (a value is assigned), it will automatically be exported to the environment.

FURTHER DISCUSSION Exported variables exist in the environment space of the shell process. When the shell executes a new child process (subprocess) it passes the environment to the new process. The environment consist only of variables that have been exported. By using exported variables you can pass "global" information to any subprocess. Whether the environment is used by the subprocess depends on the subprocess.

Subprocesses cannot change the environment of a parent process. They can change their environment but the changes cannot be passed back to the parent.

If **export** is executed without any arguments, it lists out all exported variables. If you are in a subshell and use **export** with no arguments, only the variables that have been exported in the subshell are listed. To list all exported variables that were passed to a subshell use the **env** command described in Module 41.

TIP: It is not unusual to have control characters assigned to variables. These can cause your output to be garbled when you display them using the **export** command. To remove or convert them for display purposes use the following command.

```
export | col
```

or

```
export | cat -v
```

RELATED COMMANDS Refer to the **env** command described in Module 41 and the **set** command described in Module 118. The csh uses setenv to perform the same function, see Module 119.

RELATED FILES The **export** command writes to the standard output when no arguments are given.

APPLICATIONS

The **export** command is used to allow information to be passed from a parent process to a child process. For example, the **vi** editor uses the environment SHELL and TERM variables to know what type of terminal it is working with and which shell to invoke when you request one.

TYPICAL OPERATION

In this activity you use the **export** command to place the COLUMNS variable in the environment. You then start a subshell and view the exported commands using **export**. If you are using the csh, skip to the next module. Begin at the shell prompt.

1. Type **export** and press **Return**. Notice all of your environment variables are displayed.
2. Type **export COLUMNS=80** and press **Return**. No response is sent to your terminal.
3. Type **export** and press **Return**. Notice now the COLUMNS variables is in the environment.

```
cj> export
COLUMNS=80
ENV=/u1/ts/mylogin/.kshrc
FCEDIT=vi
HOME=/tmp
LOGNAME=mylogin
MAIL=/usr/mail/mylogin
PATH=:/u1/ts/mylogin/bin:/bin:/usr/bin:/usr/lbin:/usr/local/bin
SHELL=/bin/ksh
TERM=adm5
TMPDIR=/tmp
TZ=CST6CDT
VISUAL=vi
```

4. Turn to Module 41 (SV), Module 118 (BSD) to continue the learning sequence.

Module 48
expr

DESCRIPTION

The external **expr** command evaluates and compares expressions. An expression is composed of operands and operators. The operands are strings or integers; the operators are special characters used to inform **expr** how to interpret and compare the operands.

Operands and operators must be separated by blanks. Characters having special meaning to the shell must be escaped using the backslash (\) or quoted by single or double quotes. The parenthesis used for regular expression must always be preceded by backslashes. Strings containing blanks must be quoted to be considered one argument. You may precede integers with a minus sign (–) to represent a negative number.

The **expr** command can be used to perform many functions but it should be used as a last resort command. The ksh and csh shell provide much faster and usually easier ways of performing calculations and comparisons.

COMMAND FORMAT Following is the general format of the **expr** command.

```
expr arg1 operator arg2 ...
```

Multiple operators may be specified on the command line.

Arguments The following list describes the arguments that may be passed to the **expr** command. The list is in decreasing precedence of the operators.

String Matching Operators

arg1 : arg2 Matching operator. Compares the characters in *arg1* with the characters in *arg2*. Returns the number of characters in *arg1* matched by characters in *arg2*. *Arg1* should be a string and *arg2* should be a regular expression recognized by the **ed** command, with the exception of the ^ character, because all expressions start at the beginning of *arg2*. The following examples illustrate the use of the matching operator:

```
$ expr string : str          # First 3 characters match
3

$ expr string : ing          # Last 3 match but comparison
0                            # must start at beginning of arg1

$ expr string : strg         # Arg2 must match arg1 in its
0                            # entirety

$ expr string : '.*'         # .* is a regular expression
6                            # that matches any number of
                             # any characters
```

```
$ expr string : '.*i'          # .*i matches any set of
4                              # characters ending with i
```

You may find times when you need to return the matching portion of the string instead of the number of matching characters. You can use the \(...\) notation to perform this function. For example:

```
$ expr string : '\(.*\)'       # Returns all chars in arg1
string

$ expr string : '..\(..\)'     # Skips first 2 chars of
ri                             # arg1, returns next 2

$ expr string : '\(...\)'      # Returns first 3 chars of
str                            # arg1

$ expr string : '.*\(...\)'    # Returns last 3 chars of
ing                            # arg1

$ expr string : '..\(.*\)..'   # Returns center of arg1
ri                             # first and last 2 chars
                               # removed

$ expr string : '\(stx\)'      # Returns nothing

$ expr string : 'st\(.*\)'     # Return all of arg1 after
ring                           # skipping st
```

Integer Arithmetic Operators The following operators perform integer arithmetic. The returned value of a division operation is truncated to the largest integer. The multiplication, division, and modulus functions have equal precedence. The addition and subtraction operators have equal precedence but are lower in precedence than the previous three arithmetic operators. The * for multiplication must be escaped with a \ (backslash) or quoted to avoid shell interpretation.

*arg1 * arg2* Displays the value of *arg1* multiplied by *arg2*. For example,

```
$ expr 80 \* 24
1920
```

arg1 / arg2 Displays the integer value of *arg1* divided by *arg2*. For example,

```
$ expr 15 / 4
3
```

Notice the result of 15 / 4 is 3.75 but the value is truncated to the next lower number.

arg1 % arg2 Displays the remainder (modulus) of *arg1* divided by *arg2*. For instance,

```
$ expr 15 % 4
3
```

1*arg1 + arg2* Display the value of *arg1* added to *arg2*. For example,

```
$ NUM=0
$ expr $NUM + 1
1
```

arg1 – arg2 Display the value of *arg2* subtracted from *arg1*. For example,

```
$ expr $NUM - 1
0
```

Comparison Operators These operators are used to compare numbers or strings. A true condition displays a 1 and a false condition displays a 0. The < and > symbols must be escaped or quoted to prevent interpretation by the shell.

NOTE

The return code from **expr** equals 0 if the expression is true. If the expression is false the return code is 1. Thus if **expr** displays a 0 the return code is 1. If a 1 is displayed the return code is 0.

arg1 = arg2 Displays a 1 if *arg1* and *arg2* are equal. For example,

```
$ NUM=5
$ PREV=6
$ expr $NUM = $PREV          # NUM and PREV are not equal so
0                            # 0 is displayed.
```

arg1 \> arg2 Displays a 1 if *arg1* is greater than *arg2*.

```
$ expr dog \> cat            # Since d has a higher ASCII
1                            # value than c a 1 is displayed.
```

arg1 \>= arg2 Displays a 1 if *arg1* is greater than or equal to *arg2*.

arg1 \< arg2 Displays a 1 if *arg1* is less than *arg2*.

arg1 \<= arg2 Displays a 1 if *arg1* is less than or equal to *arg2*.

arg1 !> arg2 Displays a 1 if *arg1* is not equal to *arg2*.

Conditional Operators

arg1 \& arg2 Returns the value of *arg1* if *arg1* and *arg2* are not 0 or NUL; otherwise, 0 is returned. *Arg1* can be another expression. For instance,

```
$ VAR=""                     # Set VAR to NUL
$ PRT=""                     # Set PRT to NUL
$ expr $VAR \& $PRT          # Return 0, arg1 and arg2
0                            # are NUL

$ VAR="hplj"                 # Set VAR to hplj printer
$ PRT="hpij"                 # Set PRT to hpij printer
$ expr $VAR \& $PRT          # VAR is set, return its
hplj                         # value
```

arg1 \| arg2 Returns the value of *arg1* if it is not 0 or NUL; otherwise, the value of *arg2* is returned. *Arg1* may consist of another expression (arg1 operand arg2). The following example reveals the syntax and use of this operand.

```
$ VAR=""                     # Set variable VAR to NUL
$ expr $VAR \| undefined     # If VAR is NUL return
undefined                    # the undefined string
```

DIAGNOSTICS AND BUGS The following error messages may be returned by **expr**:

syntax error	If the operator or operand are not correct
nonnumeric argument	If a string is used in an arithmetic expression

You must be careful in passing arguments to **expr**. It does not know the difference between operands and operators except by their values. Therefore, you may have to add a character to the operands for proper interpretation. For example,

```
A="+"
B="+"
expr $A = $B
```

would cause a problem, because **expr** interprets the + signs as operators instead of operands. To rectify this problem you can add some valid character to the operands:

```
A="x+"
B="x+"
expr $A = $B
```

RELATED COMMANDS Refer to the **let** commands described in Module 73.

RELATED FILES The **expr** command writes its results to the standard output.

RETURN CODES The following exit statuses are returned by **expr**:

0	If the expression is not 0 or NUL
1	If the expression is 0 or NUL
2	If the expression is invalid

APPLICATIONS

You can use the **expr** command to perform arithmetic or string manipulation and making comparisons. It has been an old standby in the writing of shell scripts to perform numeric and string manipulation functions. With the arrival of the Korn shell and its many internal features, the **expr** command has lost ground in the number of uses it offers. The Korn shell's variable substitution and **let** command replaces many of the **expr**'s functions. Although it can still be used to perform such needs, it is extremely slow when compared to the internal Korn shell commands.

TYPICAL OPERATION

In this activity you use the **expr** command to display the numbers from 1 to 10. This will demonstrate the speed of the **expr** command. Begin at the shell prompt.

1. Type the following code at your shell prompt or enter it as a shell script. Press **Return** at the end of each line.

```
NUM=1
while [ "$NUM" -le 10 ]
 do
  echo $NUM
  NUM=`expr $NUM + 1`
 done
```

C Shell

The csh code follows.

```
set NUM=1
while ($NUM < 10)
echo $NUM
@NUM=$NUM+1
end
```

---

2. Turn to Module 59 to continue the learning sequence.

Module 49
false

DESCRIPTION

The internal **false** command always returns an exit status of one. It is used to guarantee a nonzero return code for a conditional statement. The shell considers a return code of zero as true and nonzero as false.

COMMAND FORMAT Following is the general format of the **false** command.

```
false
let 1
```

The following alias is automatically set by the Korn shell when you log in:

```
alias false='let 1'
```

FURTHER DISCUSSION The **false** command lends itself well to shell programming. It is a sure-bet command that always returns a status of one (false). Thus you can program infinite loops and break out based on various conditions.

Common code using the **false** command resembles the following:

```
FILE=$1
until false
 do
  if [ -f "$FILE" ]
  then
       echo "\007 The file $FILE has been created!"
       break
  fi
  sleep 5
 done
```

This code will run until the FILE specified by the first parameter is created. When the file is created, the echo line is displayed and the loop is exited via the **break** command.

NOTE

The **false** command returns a 1. The shell uses 0 as a true condition. Most programming languages, such as C, use 1 or nonzero numbers as true conditions. The reason for this convention is that return codes from programs are 0 if the program completed successfully and nonzero if a problem occurred.

RELATED COMMANDS Refer to the **true** command described in Module 142.

RETURN CODES The **false** command always returns a nonzero return code.

APPLICATIONS

The most common use of the **false** command is in looping conventions. You can use a **false** command in an **until** loop statement to create an infinite (endless) loop. Some other condition, such as a **case** or **if** command, must execute a **break** or **exit** to leave the loop. The **false** command may also be useful in testing shell scripts. You can place the **false** command on **if** statements and **while** or **until** loops to test how a shell script performs in these conditions.

TYPICAL OPERATION

In this activity you use the **false** command to loop indefinitely while printing the time to your screen. Begin at the shell prompt.

C Shell

If you are using the C shell enter the shell by typing **sh** and pressing **Return**.

---

1. Use the **vi** editor to create the following shell script. Type **vi clock** and press **Return** to enter the editor. After you have typed in the shell script code, exit **vi** by typing **ZZ**.

```
CLR=`tput clear`                    # set CLR variable to clear sequence
 until false                        # until loop return code = 0
  do
   echo "${CLR}\n\n\n"             # CLeaR the screen and print new-lines
   banner `date +%H:%M`             # Display time
   sleep 59                         # Sleep for 59 seconds
  done
```

BSD (Berkeley)

```
CLR=`clear`                         # set CLR variable to clear sequence
until false                         # until loop return code = 0
 do
  echo "${CLR}
"                                   # CLeaR the screen and print new-lines
echo "                  "`date`     # Display time
sleep 59                            # Sleep for 59 seconds
done
```

This shell will display the hours and minutes (date +%H:%M) in large letters (banner) in the center of your screen. BSD does not support the **banner** command; therefore, remove it.

2. Change the modes to executable for the script by typing **chmod 777 clock** and pressing **Return**.

3. To run your new clock command type **clock** and press **Return**.
4. To stop your clock press **Delete**.

C Shell

If you entered sh press **Ctrl-D** to return to your csh.

---

5. Turn to Module 135 to continue the learning sequence.

Module 50
fc (ksh) / ! (csh)

DESCRIPTION

The internal **fc** command provides access to your history file. The Korn shell creates a history file containing the commands you have previously entered during your current login session. You use the **fc** utility to:

- Display the contents of the history file.
- Edit (Visually) the contents of the history file.
- Reexecute previous commands stored in the history file.

C Shell

The **csh** supports history file manipulation with different syntax and functionality. There is no command like **fc** in **csh**; instead, a character is used to access the history file. The default character is an exclamation mark (!). You can reset this using the **histchar** variable. Unlike the **fc** command in the Korn shell the C shell does not support visual editing of the history file.

COMMAND FORMAT Following is the general format of the **fc** command.

```
fc [ -e editor ] [ -lnr ] [ first [ last ] ]
fc -e - [ old=new ] [ command ]
```

history is an alias for **fc -l**.

r is an alias for **fc -e -**. The **r** command is called the repeat command. It searches for a string in the history file that matches the given argument and then executes the entire line in which the string was found.

C Shell

```
!reference_string
^l^r[^]
```

The **!** mark signals the shell that you wish to reexecute a command from the history file. The *reference_string* is composed of three parts. They are the event separator, word designator, and modifiers. These are described in the following sections. The command lines stored in the history file are often referred to as *events*.

The second format is referred to as *Quick Substitution*. It is equivalent to the history substitution command,

```
!:s^l^r[^]
```

which substitutes string *l* with string *r* on the previous command line and executes the new line if the substitution was successful.

Options The following list describes the options and their arguments that may be used to control how **fc** functions.

First format

-e *editor* — Use editor *editor* to edit the commands in the history file. The editing of the history file is referred to as in-line command editing. If you do not specify the -e option, **fc** checks the value of the FCEDIT variable for an editor name. If FCEDIT is not defined then /bin/ed is used. Valid editors are **vi**, **emacs**, **gmacs**, and **ed**.

-l — List the contents of the history file.

-n — Suppress the command numbers when displaying the commands from the history file.

-r — Reverse the order of the commands in the history file when they are listed.

-first — Beginning command. The number of lines to go back from the last command to start listing out the history file. For example,

```
fc -l -25
history -25
```

lists the last 25 commands saved in the history file.

-last — Ending command. The number of lines to go back from the last command to end listing out the history file. For example,

```
fc -l -n -25 -5
history -n -25 -5
```

lists out the last commands starting 25 commands ago down to 5 commands ago without command numbers.

Second Format

-e - — Reexecutes the previous command in the history file. If you do not specify a command or partial command, then the last command you executed is re-executed. This command is the **r** alias (**alias r="fc -e -"**).

Arguments The following list describes the arguments that may be passed to the **fc** command.

First Format

first — List commands starting at line *first* in the history file. For instance,

```
fc -l 15
history 15
```

displays the commands beginning at line 15 to the end of the history file.

last List commands up to line *last* in the history file. For example,

```
fc -l 20 30
history 20 30
```

displays lines 20 through 30 of the history file.

Second Format

old=new Substitutes *old* string for *new* string in the appropriate *command* before reexecuting it.

command The command to reexecute from the history file. *Command* may be one of the following:

- Positive number denoting the event number associated with a command.
- Negative number subtracted from the current command number. The event number of the subtraction is executed.
- A string. The latest command starting with the string is executed. For instance,

  ```
  fc -e c
  ```

  or

  ```
  r c
  ```

  searches the history file for the last command that began with a c.
- If no *command* is specified then the previous command is executed.

Variables There are two variables that control the location and size of the history file. The HISTFILE variable informs the shell where to locate the history file. For example,

```
HISTFILE=$HOME/.history
```

places the history file in your HOME directory in the .history file. If you do not specify a path in the HISTFILE variable, the default for the location is $HOME/.sh_history.

The HISTSIZE variable is used to set the number of commands that are stored in the history file. If you do not set the HISTSIZE variable to some number it defaults to 128. Therefore, the history file will contain the last 128 commands you have entered.

The FCEDIT variable is used to define the editor used to edit your history file. The possible editors are **vi, emacs, gmacs,** and **ed.**

C Shell

The C shell uses different variables than the Korn shell. To control the size of the file, you use the **history** variable. For example,

```
set history=100
```

would cause the shell to save the last 100 commands in the history file.

The **savehist** variable is available in the csh to save *n* commands in your history file when you log out. The next time you log back in to the system, those *n* commands are still available in your history file. If you do not set **savehist** to a number of history commands to save, your history file is recreated each time you log in.

You can set the **savehist** variable by placing the **set savehist=50** command in your .login file. This will save the last 50 commands in your history file when you log out of the system. The next time you log in the same last 50 commands will be available in your history file.

Another variable may be set to change the history access characters. By default you type ! or ^ to access a previous command residing in the history file. You can change these characters to alternative characters by setting the **histchar** variable. For example,

```
set histchar=.,
```

changes your history substitute characters to a period (.) and a comma (,). Now you type a .34 to execute the 34th command in your history file. To perform quick substitution on the last command executed use the , notation instead of the caret (^). For example, type ,-xa,-la, to change xa to la in your last command.

If the **verbose** variable is set, each command is displayed after history substitution is performed. You set this variable using the -v option of the **csh**.

IN-LINE EDITING In-line editing allows you to make corrections to a command you are currently typing or to a previous command stored in the history file. To activate in-line editing you must have the EDITOR variable set to **vi** or **emacs**. You may use the set -o **vi** or set -o **emacs** command to select an in-line edit mode. This informs the Korn shell you want to be in in-line edit mode on each command line.

Once the in-line editing mode is activated you can edit your commands and resend them to UNIX by typing a Return. The Return must be a Ctrl-J character for some shells to send the command line to UNIX.

The window is a single line that can be scrolled left or right. If text is scrolled off on the left of the screen a > appears. If text is scrolled off on the right side a < appears. The width of the screen can be set by using the COLUMNS variable. The default for COLUMNS is 80.

Vi Editing Mode To use the **vi** edit command in your history file you must first have the FCEDIT and EDITOR variables set to **vi**. To edit commands saved in your history file simply press the Escape key. You are now in **vi** command mode. If you want to make a correction to the previous command simply press **k**. Now move around on the line using **vi** commands. Make changes to the line then press Return. You don't have to press Escape before pressing Return.

The **vi** editing mode is the same as using the **vi** editor, with a few limitations. The window is only one line and cannot be expanded. Some of the standard **vi** commands are not supported because of this reduced window size. The following is a list of commands NOT supported by in-line **vi** editing mode. Refer to Module 151 for available vi commands.

o	Open a line below for inserting text.
O	Open a line above for inserting text.
scrolling	Screen scrolling (Ctrl-U, Ctrl-D, Ctrl-E, Ctrl-Y).
macros	Storing editor commands in registers.
mapping	Mapping a character to perform multiple edit commands.
:	Ex mode. You cannot use any colon commands.

The following commands are supported in in-line **vi** edit mode but not in the **vi** editor.

*n*_	Append the *n*th argument **from the previous** command line after the cursor and enter insert mode.

word* Replace the current *word* with the output of filename generation from the *word**. If no match, the word is left unchanged. If expansion occurs, then the cursor is moved to the end of the text and input mode is entered.

v This causes the in-line editor to enter the editor specified by the VISUAL variable if it is set. If it is not, then the value of the EDITOR variable is used. You can make corrections to the command line, exit the editor, and the shell reexecutes the command automatically. It is extremely useful for editing multiline command lines. This is equivalent to typing fc -e vi or just fc if the FCEDIT variable is set to vi.

For example, assume you have typed a while loop structure as follows on the command line.

```
while :
 do
   if [ -f anewfile ]
   then echo "File is here."
        braek
   else sleep 15
   fi
 done
```

Notice the break command is misspelled. You have executed the command and realize the problem. To correct the error simply press Delete to stop the execution of the loop. Press Escape kv to enter the editor. The Escape enters the history editing feature. The k moves up one line and the v enters you into the editor. Now make the necessary changes (/braek<Return>llxp) and exit the editor by typing ZZ. The command is reexecuted by the shell immediately. The command is displayed on your screen and executed.

= Pathname listing. Causes the current word on the command line to be expanded to a list of pathnames. For example,

```
$ls m<ESC>=
1) myfile
2) myhome
$
```

\ Pathname completion. Causes the current word on the command line to be expanded to a filename or directory. If it is a directory then a / is added. For example,

```
$ echo my<Esc>\
```

would expand to myfile.

@letter Searches your list of aliases for one named *_letter*. If an alias is defined, its value is inserted as keystrokes. For example,

```
alias _Q='LBi"^V<Esc>Ea"^V<Esc>'
```

now if you are editing a command and type <Esc>@Q the current word you are on would be quoted.

Insert a # (comment marker) at the beginning of a command.

The **vi** editor is described in Module 151.

Emacs Editing Mode The emacs editing mode is based on the **emacs** editor. The following list describes the emacs commands. The *n* denotes a number that may be entered to provide a count for the command.

Cursor Movement

*n*Ctrl-B	Move cursor back *n* characters.
*n*Ctrl-F	Move cursor forward *n* characters.
Ctrl-A	Move cursor to beginning of line.
Ctrl-E	Move cursor to end of line.
*n*Esc-b	Move cursor back to beginning of *n*th previous word.
*n*Esc-f	Move cursor forward to beginning of *n*th word.
Ctrl-]*c*	Move cursor to next character *c*.
Esc-space Ctrl-@	Set mark at cursor position.
Ctrl-X Ctrl-X	Exchange cursor position and mark.

Line Search

*n*Ctrl-P	Select *n*th previous command line of history file.
*n*Ctrl-N	Select *n*th next command line.
Esc-<	Select oldest command line.
Esc->	Select newest command line.
*n*Ctrl-R*str*Return	Search for the *n*th command line matching *str*.

Changing Text

*n*Ctrl-H	Delete *n* characters back from cursor.
*n*Ctrl-D	Delete *n* characters forward from cursor.
*n*Esc-h	Delete *n* previous words.
*n*Esc-d	Delete *n* words.
*n*Ctrl-K	Delete to end of line.
Ctrl-G	Delete entire line.
Ctrl-W	Delete from cursor to mark.
Ctrl-T	Transpose characters.
Ctrl-C	Capitalize character.
Esc-c	Capitalize word.
Esc-l	Lowercase word.
*n*Esc-. *n*Esc-_	Insert *n*th word of previous command.
Esc-Esc Esc-*	Replace current word with files that would match the word followed by a *.

Miscellaneous

Ctrl-J Ctrl-M Return	Execute command line.
Ctrl-L	Redisplay current command line.
Ctrl-O	Remember next command line, execute current line, then recall next line to window.
Ctrl-U	Multiple next count *n* by 4.
Ctrl-V	Display version of the shell.
\	Do not interpret next character. Allows insertion of Control and Escape characters in command lines.
Ctrl-D	Terminate the shell, log out.
Esc-*n*	Enter count for following command.

C Shell History File Editing

The C Shell uses a line editor approach to edit the history file. That is, you type in a command and the shell locates the command line in the history file and performs the editing commands you specified. Each command line in the history file is referenced by an event number. For example, the following output is from the **history** command:

```
1    mail dwh
2    write skb
3    vi wsr.092989
4    find /usr/local -name "dly.*" -print
5    vi apiprog.c
```

Event number 1 references the command line "mail dwh." The command,

```
!3
```

reexecutes the "vi wsr.092989" command.

Event Designators An event designator references a command line stored in the history file. There are several formats of event designators, they are as follows.

!	Start a history substitution command, except when followed by a space, tab, new-line, =, or (.
!!	Reference the previous command line. Typing **!!** and pressing Return executes the previous command.
!*n*	Reference command line with event number *n* from the history file. This references a specific event number. For example, if the current event number is 7, you can specify an event number from 1 to 6. The command **!3** would reexecute event 3, **vi wsr.092989**.
!-*n*	Reference the *n*th previous command event. The current event number -*n*. For example, if the current event number is 7 and you type **!-3**, event number 4 is executed.
!*str*	Reference the most recent command line starting with string *str*. For example, **!vi** reexecutes the "vi apiprog.c" command.
!?*str*[?]	Reference the most recent command line containing the string *str*. For example, **!?dly** reexecutes the "find ..." command.
!{*str*}	Insulate a reference string from adjacent characters.

Word Designators Word designators provide a way to reference specific arguments (words) from a command line stored in the history file. A colon (:) separates the event designator and the word designator. If the word designator begins with $, %, ^, *, or - the : is not needed. If you want the word to be selected from the previous command line, the second ! is not required. For example, **!!:2** and **!:2** both refer to the second word on the previous command. Words are zero relative, that is, word zero is the command itself. The first argument (arg1) on the command line is word 1.

The available word designators are as follows:

#	The entire command line.
0	The first word on the command line, argument 0, which is the command itself. For example, **!2:0** refers to the word "write" in the previous history listing.

$	The last argument on the command line. For example, **!4:$** refers to the word "-print".
%	The word matched by the most recent **?s** event designator.
^	The first argument (second word) on the command line. For example, **!4:^** refers to "/usr/local" on the "find ..." command line.
n	The *n*th (*n*th + 1 word) on the command line.
*	All arguments on the command line. Does not include word 0. This is the same as 1-$.
x-y	A range of words starting at word *x* and continuing through word *y*.
*x**	An abbreviated form of *x*-$, which refers to word *x* through the last word.
x-	The same as *x** except the last word is not included.
-y	Same as the range 0-*y*.

Modifiers You can add modifiers after the word designators. A modifier is used to modify the referenced command line. Each modifier must be preceded with a colon (:).

&	Repeat the previous substitution(s/*old*/*new*/).
e	Remove everything but the suffix.
g	Globally apply the change to the first occurrence of a match in each word. The g must precede other modifiers.
h	Remove the last filename or directory name from the path. Same as performing a dirname on the designated word.
p	Print (display) the command but do not execute it.
q	Quote the substituted words, preventing further substitutions.
r	Remove a suffix of the form ".xxx," leaving the basename.
s/*old*/*new*/	Substitute the *new* string in place of *old* string.
t	Remove all leading pathname components, leaving the basename.
x	Same as q, except words are delimited by spaces, tabs, and new-lines.

History Editing Examples The following examples illustrate how to use the Event Designators, Word Designators, and the Modifiers.

history	List out the history events.
!10	Execute history event number 10.
!{l}.c	Search for first event beginning with l and add .c to the end of the command line.
!vi	Execute the last command that started with vi.
^bni^bin	Substitute "bni" in the previous command line to "bin."
!vi:s/1.c/2.c/	Substitute "1.c" to "2.c" on the last command that started with vi.

RELATED COMMANDS Refer to the **history** command described in Module 64.

RELATED FILES The **fc** command searches, displays, and edits your **ksh** history file. The **!** (exclamation mark) signals the **csh** to access your history file as requested.

APPLICATIONS

You use the **fc** command or an alias thereof to execute previous commands stored in the history file. The commands can be referred to by string or event numbers. This allows you a quick and easy way to reexecute a command without retyping it.

In-line editing lets you edit a previous command before executing it. This provides flexibility in accessing the history file and saves you even more time from retyping commands.

C Shell

The **csh** provides similar functionality with the **!** notation. You use the **!** to access previous commands stored in the history file.

TYPICAL OPERATION

In this activity you use the **fc** (**ksh** only) command to list the last commands from the history file. Then you execute a command beginning with a specified letter. Begin at the shell prompt.

1. Type **fc -l** and press **Return**. Notice the last 16 (default value) lines of your history file are displayed with event numbers. The **history** alias command performs the same function.

C Shell

Type **history** and press **Return**.

---

2. Press **<Esc>** and type **/cat** and press **Return**. Notice that the last cat command you execute is displayed. To execute the cat command press **Return** or type **n** to search for the next cat command.

C Shell

Type **!cat** and press **Return** to execute the last cat command.

---

3. Type **r** and press **Return**. The last command that began with a c is executed.

C Shell

Type **!c** and press **Return**. The last command that began with a c is executed.

---

4. Press **<Esc>** and type **kv** and press **Return**. You are now in the vi editor. Change the name of the file to another filename. Type **wc$/etc/motd** and press **<Esc>**. Now type **ZZ** to exit vi and run the command.
5. Turn to Module 62 to continue the learning sequence.

Module 51
fg (csh, ksh)

DESCRIPTION

The internal **fg** command resumes stopped or suspended jobs by placing them in the foreground. The **fg** command is part of the job control feature of the C shell and the Korn shell. If a job is in a suspended state you can no longer interface with it. When a job is brought back into the foreground it becomes interactive again.

COMMAND FORMAT Following is the general format of the **fg** command.

```
fg [ %job ... ]
```

Arguments The following arguments may be passed to the **fg** command.

%job A job number associated with a process. If the interactive option is turned on, *job* may be one of the following:

%num	A job number associated with a process
%string	The job whose name begins with *string*
%?string	The job whose name contains *string*
%+ *%%*	Current process, the last process you stopped or suspended
%-	Previous process, the job before the current job

RELATED COMMANDS Refer to the **bg**, **jobs**, and **kill** commands described in modules 9, 68, and 70.

APPLICATIONS

The **fg** command is used to force suspended or stopped jobs into the foreground. This brings the job back into interactive mode, allowing you to interface with it from your keyboard. For instance, if you are in **vi** and decide to suspend the editor, you type Ctrl-Z. Now **vi** is suspended in background. You can use the **jobs** command to display all background and suspended jobs. After you perform various other commands you can return to your suspended **vi** session by typing **fg %1** and pressing **Return**. Since this is the current job you could just type **fg**.

TYPICAL OPERATION

In this activity you use the **fg** command to restart a process you have suspended. Begin at the shell prompt.

1. Type **vi x** and press **Return**.

   ```
   cj> vi x
   ```

2. Press **Ctrl-Z** to suspend **vi**.

   ```
   ^Z
   ~
   .
   .
   ~
   "x" [New file]


   [1] + Stopped                 vi x
   cj>
   ```

   The [1] specifies the job number.

3. Type **ps** and press **Return**. This shows the **vi** process in your process table. You can continue to perform commands until you are ready to reenter vi.
4. Type **fg %1** and press **Return** to restart your **vi** session. You will return to your **vi** session.
5. Type **:q!** and press **Return** to exit **vi**.
6. Turn to Module 68 to continue the learning sequence.

Module 52
fgrep

DESCRIPTION

The external **fgrep** command is used to search ASCII text files for a specific character string. Any lines containing the string are written to the standard output. The pattern may be a word or string of characters. Regular expressions are not supported by the fgrep command. **fgrep** is designed to be a fast, simple form of the **grep** command. If **fgrep** searches more than one file, each displayed line is preceded by the filename. If no filenames are given, **fgrep** reads from the standard input. Thus you can use **fgrep** in a pipe command.

The name **fgrep** is a combination of editor command characters. It is from the editor command :g/RE/p, which translates to global Regular Expression print. The f stands for fast. Thus fgrep is fast global RE print. Unfortunately, this is a misnomer because **fgrep** cannot search for Regular Expression patterns.

NOTE

> The **fgrep** command performs line oriented searches. Therefore, a phrase that spans more than one line is not matched. Thus when deciding what string to use for the search, you should try to keep it as short as possible yet as unique as possible.

COMMAND FORMAT Following is the general format of the **fgrep** command.

```
fgrep [ -bchilnvx ] [ -e -string ] string  file_list
fgrep [ -bchilnvx ] [ -e -string ] -f file file_list
```

BSD (Berkeley)

```
fgrep [ -bcilnvx ] [ -e -string ] string  file_list
fgrep [ -bcilnvx ] [ -e -string ] -f file file_list
```

The first form of the command uses the *string* provided on the command line to search the file_list for a match. In the second form, multiple *strings* are stored in *file* and **fgrep** searches the *file_list* for each *string* listed.

Options The following list describes the options and their arguments that may be used to control how **fgrep** functions.

-b Displays the block number in which the *pattern* was found before the line that contains the matching *pattern*.

-c Displays only a total count of matching lines for each file processed.

-e *-string* — The *string* begins with a dash. This allows you to specify a string that begins with a dash. Normally, any argument beginning with a dash is interpreted as an option, not a string or argument.

-f *file* — Read in the strings to search for from *file*. This allows you to create a file containing all of the strings you want **fgrep** to search for in the file_list or standard input.

-h — Suppress the displaying of filenames which precede lines that match the specified patterns when multiple files are searched.

-i — Ignore the difference between uppercase and lowercase characters during comparisons.

-l — Displays only the names of the files containing the specified *pattern*. The lines containing the *patterns* are not displayed.

-n — Displays the line number before each line containing the *pattern*.

-v — Displays only the lines that do not match the *pattern*. The v command in the **ex** editor performs the same type of function. It is an exception search. Search for every line except the ones containing the given *pattern*.

-x — Displays only those lines matched in their entirety.

Arguments The following list describes the arguments that may be passed to the **fgrep** command.

string — Any combination of characters that are literally interpreted by **fgrep** as strings to search for in the given input stream of data

file_list — One or more files to search for the given *pattern*

SEARCH PATTERNS The **fgrep** command differs from the **grep** and **egrep** commands in that it does not support Regular Expression matching. It is designed to perform fast, simple string searches. It is normally used when you know the exact string you want and a fast search time is important.

DIAGNOSTICS AND BUGS The input lines read by **fgrep** are limited to the system's BUFSIZ definition. If a line is longer than BUFSIZ characters, the line is truncated before the comparison is made. The BUFSIZ used by your system is normally found in the /usr/include/stdio.h file.

BSD (Berkeley)

Input lines are limited to 256 characters. Lines longer than 256 characters are truncated.

RELATED COMMANDS Refer to the **nawk**, **ed**, **ex**, **egrep**, **grep**, **sed**, and **ksh** commands described in modules 6, 39, 43, 40, 60, 117, and 71.

RELATED FILES The **fgrep** command can read from the standard input or a specified list of files. It writes to the standard output. It can also read multiple search strings from a specified file using the -f option.

RETURN CODES The return code is set to 0 if any *patterns* were found. If no *patterns* were found, the return code is 1. If syntax errors are present in the *pattern* or if files are not accessible, a return code of 2 is returned, even if *patterns* were found.

APPLICATIONS

You can use **fgrep** to search text for a specific word or string of characters. The text may be contained in a file or be from the standard input. The various uses of grep include finding lines or files that contain a specific *string*, counting the number of lines that contain the *string* in a file, showing the line numbers of lines that contain the *string*, and showing the block that contains the *string*. These uses may be inverted to function on all lines that do not contain the specified *string* also.

The uses of **fgrep** are far reaching, from just searching a file for lines that contain a *string* to using **fgrep** inside shell scripts to verify a *pattern* exists. It is one of those commands that the more you use it the more you can't seem to live without it.

The difference with **fgrep** and the other grep commands is its search capabilities and speed. It should be used in situations where you have a large file to search and you know the string you want to match.

TYPICAL OPERATION

In this activity you use the **fgrep** command to locate your system's BUFSIZ in the stdio.h file. An example using the -f option is also included to show how to store search strings in a file. Begin at the shell prompt.

1. Type **fgrep -i bufsiz /usr/include/stdio.h** and press **Return**. This returns the maximum number of characters an input line can contain that **fgrep** can search for a string match. Notice the case of bufsiz; the -i informs **fgrep** to ignore letter case.
2. Create the following file using the **cat** command. First type **cat > fstrings** and press **Return**. You may want to change the four lines listed to logins you know exist on your system. After you type one *string* per line press **Ctrl-D** at the beginning of the line to terminate the **cat** program.

```
cj> cat > fstrings
mylogin
luwis
bill
nancy
^D
```

3. Type **who | fgrep -f fstrings** and press **Return**. If any of the users listed in the *fstring* file are logged in to the system, they will be displayed on your screen.
4. Type **rm fstrings** to remove the *fstrings* file.
5. Turn to Module 55 to continue the learning sequence.

Module 53
file

DESCRIPTION

The **file** command determines what type of data a file contains. It returns a brief explanation of what type of data it believes is in each file.

COMMAND FORMAT Following are the general formats of the **file** command.

```
file [ -h ] [ -m magic_file ] file_list
file [ -h ] [ -m magic_file ] -f list_file
file -c [ -m magic_file ]
```

BSD (Berkeley)

file *filename*

Options The following list describes the options and their arguments that may be used to control how **file** functions.

-c — Check the magic file for format errors. Usually not used unless you are writing a new magic file. No file type determination is performed if -c is specified.

-f *list_file* — The next argument after the -f is read and the contents are used as the pathnames for files to be examined by **file**.

-h — Do not follow symbolic links.

-m *magic_file* — The next argument after the -m is used as the magic file instead of the default /etc/magic file.

BSD (Berkeley)

Does not support any options.

Arguments The following list describes the arguments passed to the **file** command.

file_list — One or more files to be checked by **file** to determine the type of data stored in them.

list_file — The file containing the pathnames of the files to be checked by **file**.

magic_file — The file to be used in place of the /etc/magic file. It is a good idea to use the -c option on this file until you remove all format errors.

BSD (Berkeley)

Does not support *list_files* or *magic_files*.

FURTHER DISCUSSION The **file** command is useful for knowing what type of data a file contains. It can help prevent you from locking up your terminal by informing you that a file contains executable program text. If you **cat** an executable file, your terminal may act strange and, in fact, the keyboard often becomes disabled. The only way out of the situation is to turn your terminal off and back on, then log back in to the system. If you perform **file** before you **cat** a file, you will know what type of data it contains and whether or not you can display it.

The **file** command makes calculated assumptions as to what type of data is in a file. The following list describes the types **file** may return. Keep in mind **file** is trying to guess what type of data is in a file. The known system types of data, such as executable or linkable object code, are known but some shell scripts and English text files may confuse **file** and thus an incorrect response is returned.

Type of data	*Description*
MACHID executable	Compiled code ready to execute
ASCII text	ASCII characters
c program text	C source code
commands text	Shell script language
English text	Plain English text
Fortran text	Fortran source code
BASIC program text	BASIC source code
COBOL program text	COBOL source code
directory	A directory file
empty	No data exist in the file

The description given by the **file** command you are using may vary but it should inform you of the same type of data stored in the file. You can display the /etc/magic file to see the definitions used by your system's **file** command.

CAUTION

The **file** command is not always correct in its assumption of the type of data stored in a file. The most common mistakes of **file** are the ASCII text, English text, and commands text.

RELATED FILES The /etc/magic file is used as a cross reference to determine what type of data is stored in a file. The magic number is based on numeric or string constants stored within the file itself. The format of the /etc/magic file is described in the first lines of the file (/etc/magic).

The **file** command writes to the standard output. Its output may be piped to another command or redirected to a file.

APPLICATIONS

Use the **file** command to determine what type of data a file contains. You can then classify the file and view it using the **cat**, **pg**, or **vi** command if it is not an executable or data file.

TYPICAL OPERATION

In this activity you use the **file** command to view the different types of files in the /etc directory. Begin at the shell prompt.

1. Type **file /etc/*** and press **Return**. Notice each file is displayed with a brief description to the right.

BSD (Berkeley)

Since **file** cannot function on multiple files you must use a loop to check each file. Some BSD systems have changed **file** so it does support multiple filenames on the command line.

```
cj> foreach F (/etc/*)
file $F
end
```

---

2. To see what your directory contains for files type **file *** and press **Return**. Your screen should resemble the following list.

```
cj> file *
bin:     directory
calendar:     ascii text
db:      directory
file1:   English text
letters:      directory
myfile: empty
```

BSD (Berkeley)

Must use a foreach F (*) loop.

---

3. Turn to Module 13 to continue the learning sequence.

Module 54
FILENAMES AND FILENAME GENERATION

DESCRIPTION

This module describes filenames and special characters used for filename generation. It is important to understand file naming conventions and how to use them before learning filename generation techniques.

A filename is a string of characters representing an *ordinary* file, a *directory* file, or a *special* file. No matter what type of file is referenced, a filename will be used to access the file.

When naming files, use meaningful filenames. It is important to know what is in each file. By using meaningful filenames you will be able to select the file you need by viewing a list of filenames. Unfortunately, the more files you have the more difficult it becomes to keep these filenames meaningful.

At some point you will want to make subdirectories that have meaningful names. You can store related files in the proper subdirectories. It is advisable to do this when you first begin using UNIX. If you plan on having a large number of files that are related, you may want to write shell scripts that control the files for you.

For example, you name all files with a three-digit filename. Inside each file place a unique string, such as @(#), followed by a description of the file. Then use the **grep** command to generate a listing of the filenames and their related description. If you use the @(#) string, the **what** command will perform this listing function for you.

The UNIX system has *de facto* standards for *special* files and has many commands you should be aware of before naming one of your shell scripts the same name. To check for existing programs use the **whence** or **find** command as follows:

```
cj> whence command          # see if command exists in your PATH

cj> find / -name file -print > TMP &   # search entire system for file
```

The use of the **find** command is not advisable on systems with large amounts of disk.

Filename Extensions The UNIX system also has *de facto* standards for filename extensions. Most extensions are one character long. Although the period (.) does not have any special meaning in a filename, except in the first position, it is used for convention. These are not hard and fast rules of UNIX, although some programs require the proper extensions. The following list provides a brief description of each extension.

Filename Extensions

.a	Assembler progamming language source code file
.c	C programming language source code file
.f	Fortran progamming language source code file
.o	A compiled object file

.h	C programming language header file
.sh	Shell programming language source code file
.z	A packed file

File Naming Rules You may name your directories and files any name you wish as long as you remember the following rules.

1. The name must be from one to 256 characters. Extra characters are ignored by the system. Keep filenames meaningful but short.
2. All characters are legal except for the slash (/). It is used to separate directory levels and files.
3. It is advisable to avoid the following list of characters since they are interpreted by the shell and other programs:

   @ # $ ^ & * ? ( ) [ ] { } / \ | ; ' " < >

   Space Tab Esc Ctrl-Characters

   Spaces and tabs must be quoted if used as part of filenames.
4. A period (.) at the beginning of a filename hides it from normal **ls** type referencing.
5. Uppercase and lowercase characters are interpreted as different characters. A file named TMP is different from a file named tmp.
6. It is advisable not to use any of the following characters at the beginning of a filename:

   + - = _

NOTE

Releases of System V prior to Release 4 only support filenames of 14 characters.

FILENAME GENERATION *Filename generation* provides a shorthand for referencing files and directories. The shell scans each *word* of the command line for filename generation characters. For every one of these characters the shell attempts to expand it to a list of matching file and directory names based on the criteria in the following wildcard section. If no matches are found then no expansion of the word is done.

Wildcards The *wildcards* used in filename generation function much like wildcards in a poker game. In poker, if a card is wild then it may be used as any other card in the deck. If "deuces" are wild then a player can make three of a kind by having a deuce and two Aces. The one difference in the shell's wildcards is the * (asterisk); it is used to expand into zero or more characters instead of a one-to-one match. The following is a list of patterns (wildcards) that may be used for matching file and directory names.

Wildcards	
*	Matches any string with zero or more characters.
?	Matches any single character.
[*chars*]	Matches any one of the characters enclosed within the brackets. The c1-c2 notation will match characters c1 through c2.
[!chars]	Matches any one of the characters *not* enclosed within the brackets.
?(*pattern*[|*pattern*...])	Optionally matches any *pattern*. Multiple patterns may be listed, separated with vertical bars (|). The brackets [ ] show optional patterns.

*(*pattern*[|*pattern*...])*	Matches zero or more occurrences of any *pattern*.
+(*pattern*[|*pattern*...])	Matches one or more occurrences of any *pattern*.
!(*pattern*[|*pattern*...])	Matches everything except any *pattern*.
@(*pattern*[|*pattern*...])	Matches exactly one occurrence of any *pattern*.
^[*user*]	Expands to value of $HOME. If *user* is specified expands to user's home directory.

C Shell

The csh does not support the ?(pattern), *(pattern), +(pattern), !(pattern), or @(pattern) formats. However, the C shell adds the notation of {literal1,literal2,..}. You can use this notation to match various literal strings within filenames. The matching is performed left-to-right, with the order of the output being sorted for each group of files that match. It becomes useful when you want to match several different strings within filenames.

The following examples provide insight into how filename expansion works.

```
cj> ls *.db              # list all files ending in .db.
cj> ls [0-9]*            # list all files beginning with a number between 0
                           and 9.
cj> echo [!0-9]*         # display all files not beginning with a number.
cj> ls *.[chza]          # display all files ending in c, h, z, or a.
cj> ls !(m*)             # list all files not beginning with m.
cj> ls +(s*)             # list all files beginning with s if more than one
                           file exists
cj> ls file*([123])      # list file followed by any number of digits from
                           1 to 3.
cj> ls file@([123])      # list file1, file2, and file3.
```

C Shell

```
cj> ls api{327,525}0.c # list all files beginning with api and
                       # ending with 0.c that have 327 or 525 in
                       # between.
```

NOTE

If too many matches are found, the shell will expand the filenames and the command will not be able to handle all of the arguments. The command will abort with a message saying "too many arguments" or "stack overflow" or "out of space."

Literals The literals used in filename generation must be matched exactly. Files beginning with . (period) must be matched by using a period to begin the reference to the filename. Matching the / (slash) in directories must also be done by using the slash. The following table explains the use of literals in filename generation.

Literals	
.file	Match a hidden file; file may be composed of filename generation characters.
d/d/f	Match directories (d) and files (f) in a path. Each directory and file may contain filename generation characters.
/	Must be used to match all / characters in a pathname.

The following examples help explain the use of literals in filenames.

```
cj> ls *.db          # list all hidden files ending in db.

cj> ls */*           # list all files contained in subdirectories.
                     # does not match any files in current directory.
```

NOTE

To perform filename expansion for directories, you must specify a match for each level of directories. The */*/* command will not match files in the */* directory.

FILENAME COMPLETION The **ksh** shell provides a filename completion feature. It is performed by the in-line editor. If you are using the **vi** in-line editor, simply type your command and part of your filename. Press Escape and type an *. The shell expands the partial filename into all matching filenames. The cursor is moved to the end of the line and you are placed back in insert mode on the command line. For example,

```
cj> ls m<Escape>*
```

generates

```
cj> ls memos misc myph_
```

C Shell

The C shell provides a filename completion feature. This feature is enabled by setting the **filec** variable. Refer to the **set** command for information on setting variables. When filename completion is enabled the **csh** completes filenames and user names during interactive sessions. The partial filename is input from the keyboard and Escape is pressed. The csh expands the partial filename into a completely unique filename. If multiple filenames match the partial string, then the filename is expanded to the end of the common part of the filename. For example, if the current directory resembles

```
cj> ls
bin
bench1
bench20
do_list
mailbox
tester_prog1
```

and you type

```
cj> te<Esc>
```

the **csh** expands *te* to tester_prog1. But if you type

```
cj> be<Esc>
```

then the **csh** expands *be* to bench and beeps your terminal to signal you that completion is incomplete because two filenames matched *be*.

APPLICATIONS

Filenames are used to reference data stored on disk. Therefore, you should use meaningful filenames to explain the function or contents of your files.

The wildcards and literals provide for easy referencing of filenames. Wildcards and literals may be used to restrict your referencing to groups of files having common characteristics. They are often used to display, copy, delete, or rename files or groups of files.

The literals are used to reference hidden files or directories and subdirectories. They provide an extension to the files not usually listed in the current directory.

TYPICAL OPERATION

In this activity you use various UNIX commands to perform filename generation. Begin at the shell prompt in your HOME directory.

1. Type **ls -C /** and press **Return**. Notice the / (root) directory is displayed.
2. Type **ls -C /*** and press **Return**. Notice the output is different. Each directory under the / directory has been listed. This probably takes a while to list out.
3. Type **cp f* db** and press **Return** to copy all files beginning with f to the db directory.
4. Display the db directory by typing **ls db** and pressing **Return**.

   ```
   cj> ls db
   file1
   file2
   phone
   ```

5. **Turn to Module 113 to continue the learning sequence.**

Module 55
find

DESCRIPTION

The external **find** command locates files that match specified expressions. It starts searching for files in the given directory and continues its search through all subdirectories. Multiple directories may be searched by providing the name of each directory you wish searched. The expressions used by **find** may be based on the filename, file attributes, or a combination of the two.

COMMAND FORMAT Following is the general format of the **find** command.

```
find pathname_list expression_list
```

BSD (Berkeley)

```
find pathname_list expression_list
find name
```

Options The following describes the *expression_list* and their arguments you may use with the **find** command. The *expression_list* consists of one or more of the following Boolean expressions. A Boolean expression is either true or false. It the expression matches a filename or a file attribute then the expression is true; otherwise, the expression is false.

In the following descriptions the value *n* refers to an exact number. The *+n* notation refers to more than *n*, and *-n* refers to less than *n*. An *n* may be replaced with a t*u* or *-n*.

Expression_list

-atime *n* — True if the file was accessed *n* days ago. **find** updates the access times of the directories in the *pathname_list*.

-ctime *n* — True if the file's status was changed *n* days ago. A change to a file means the inode of the file was modified.

-depth — Always true if specified on the command line. Causes **find** to descend the directory structures and perform the specified options on the subdirectories and files before processing the top directory. You may find this useful when using **find** to pipe pathnames to **cpio**. If you do not have write permission in directories this will allow you to transfer the files to the archive file.

-exec *cmd* — True if the command *cmd* executes successfully and returns a zero exit status. You can use a set of braces ({}) to signify the presence of the current pathname. The *cmd* command must end with an escaped semicolon (\;). For example,

```
find . -name '*.o' -exec rm {} \;
```

would remove all files ending with a .o from the current directory structure.

-fstype *type* — True if the file system in which the file resides is of the given *type*. Valid *types* are usually **nfs** or **4.2**, where nfs refers to Network File System file systems and 4.2 refers to local BSD4.2 type file systems.

-group *name* — True if the file belongs to the group *name*. You may specify a group ID instead of a group name.

-inum *n* — True if the file has inode number *n*.

-links *n* — True if the file has *n* links.

-local — True if the specified file physically resides on the local UNIX system. This is useful when searching directories on systems that are networked together and share disk space.

-mount — True if specified on the command line. Confines the search to the file system containing the specified directory. This is useful when you only want to search one file system. If no directory is specified, the search is performed on the current directory.

-mtime *n* — True if the file's contents were modified *n* days ago.

-name *file* — True if the filename matches *file*. The shell's filename generation characters may be used if enclosed in quotes. For example,

```
find . -name '*.c' -print
```

prints all files ending with a .c in the current directory or any of its subdirectories.

-newer *file* — True if the current file is newer than the specified *file*. The modification dates are compared to decide which is newer.

-nogroup — True if the file belongs to a group (groupid) not in /etc/passwd.

-nouser — True if the file belongs to a user (userid) not in /etc/passwd.

-ok *cmd* — Same as the -exec option except the command is displayed followed by a question mark. The command *cmd* is executed only if you respond with a y.

-perm *on* — True if the permissions of the file matches the octal number *on*. Refer to Module 17 on **chmod** for a full description of the octal number representation used for *on*. You can prefix *on* with a minus sign to force the comparison of only the bits that are set in *on* to the file permission.

-print — Always true if specified on the command line. Causes the current pathname to be displayed.

-prune — Always true. Prunes the search path at the current directory level. If the current pathname is a directory, **find** will not descend into the directory.

-size *n*[c] — True if the file contains *n* blocks. If the *n* is followed by c then *n* is counted in characters instead of blocks.

-type *c* — True if the file type is type *c*, where *c* is one of the following.

Type	*Description*
b	A block special device file
c	A character special device file
d	A directory

	f	A plain file (SV only)
	p	A named pipe (FIFO) (SV only)
	f	A plain file
	l	A symbolic link to a file
	s	A socket (BSD only)

-user *name* — True if the file belongs to the user *name*. You may specify a user ID instead of a user's login name.

BSD (Berkeley)

The -mount option is not supported on most BSD systems. See the -xdev option that follows.

-ls — Always true. The full pathname and all related statistics are displayed. The statistics are:

inode number
size in kilobytes (1024 bytes)
protection mode (-rwxr-x-r-- etc.)
number of hard links
user
group
size in bytes
modification time

This is the same listing as the **ls -dgils** command. If the file is a symbolic link, the pathname of the file being linked to is displayed. A "->" precedes the linked name.

-type — Same as SV. BSD does not support the f or p type. It does support the s type for socket.

-xdev — See -mount of System V options.

Arguments The following list describes the arguments that may be passed to the **find** command.

pathname_list — This is a list of all directories to be searched by **find**. The names are separated by spaces or tabs. Each directory and all of its subdirectories are searched for matches. Pathnames may be relative pathnames. For example, to search the current directory you use the . (dot) notation.

BSD (Berkeley) Only

name — List all files whose **pathname** contains *name*. Similar to the command:

find / -name '**name**' -print

but much faster because the database is searched instead of the actual file system. Shell name generation is supported.

The database is generated from a command in root's crontab. The default setup runs the command once a week.

COMBINING EXPRESSIONS You can combine the previously defined options using the following Boolean operators.

Operator	*Description*
!	The logical NOT operator. This operator negates the expression. For example, `find . ! -name '*.c' -print` prints all pathnames NOT ending with a .c.
-a	The logical AND operator. You use this operator to combine options, causing the expression to be true only if both options or all options listed are true. It is implied in normal syntax. For example, the following two commands are equivalent. `find . -name '*.o' -size +5 -print` `find . -name '*.o' -a -size +5 -print` These commands locate files ending with .o and containing 5 or more blocks of data.
-o	The logical OR operator. You use the OR operator to cause the condition to be true if either option is true. For example, `find . -name '*.o' -o -name '*.c' -print` displays pathnames that end in .o or .c.
(*expr*)	True if the expressions enclosed in the parentheses are true. The parentheses are special to the shell and must be escaped using backslashes, as in the command `find . \( -name '*.c' -o -name '*.o' \) -print`

DIAGNOSTICS AND BUGS If you use the depth option at the root directory, **find** fails.

RELATED COMMANDS Refer to the **chmod, cpio, ksh,** and **test** commands described in modules 17, 25, 71, and 135.

RELATED FILES The **find** command reads the /etc/passwd and /etc/group files to relate user IDs and group IDs to the symbolic names you use on the command line.

RETURN CODES The return codes from **find** do not reflect whether or not files were located based on the options. Therefore, it is best to ignore the return codes.

APPLICATIONS

The **find** command serves many purposes. Its claim to fame is the ability to descend the file system tree of UNIX. It provides a way to see the contents of an entire file system or any subdirectory structure.

It is the front end processor for the **cpio** command. It generates a list of pathnames that are passed into **cpio**, which then copies the contents of each pathname to the output archive file.

It also provides the ability to perform commands with the pathnames as arguments. You can search for certain filenames and remove them if they are older than a specified amount of time. The options are many and their abilities are powerful. Although it may be hard to understand at first, it is a command well worth learning.

TYPICAL OPERATION

In this activity you use the **find** command to locate all large files in the /usr directory, old files in the current directory, and filenames with special endings. Begin at the shell prompt in your HOME directory.

1. Type **find /usr -size +500 -print** and press **Return**. The files in the /usr directory containing more than 500 blocks are displayed to your screen. The **find** command takes a while to run since it has to search the /usrfile system inode list. The output for each system will vary.

2. Type **find . -atime -2 -ok rm {} \;** and press **Return**. This will prompt you with the name of each file that has been accessed in the last 2 days. If you answer with a "y" the file is removed.

```
cj> find . -atime -2 -ok rm {} \;
< rm ... ./myfile >?  _
.
.
```

3. Type **find . \(-name '*.[achoz]' -o -name '*.bak' \) -mtime -30 -print** and press **Return**. This lists all files ending with an a, c, h, o, z, or bak and that have been modified within the last 30 days.

```
cj> find . \( -name '*.[achoz]' -o -name '*.bak' \) -mtime -7 -print
./passwd.z
```

   Your output may differ depending on what files you have in your HOME directory.

4. Turn to Module 86 to continue the learning sequence.

Module 56
finger

DESCRIPTION

The external **finger** command displays a detailed list of user information. If you specify a user_name, information for only that user is displayed. If no user_name is given, information for all users currently logged in to the system is displayed. By default, only a one-line summary of information is displayed for each user.

The following information is displayed by default:

- Login name (user_name)
- Full name, field five of the /etc/passwd file
- Terminal name (tty device)
- Write status of terminal. Terminal name is preceded by a "*" if write permission is denied to other users.
- Idle time since the last keystrokes. A single digit represents minutes, a colon-separated number is hours:minutes, and a number preceded by "d" refers to days.
- Login time
- Office location
- Phone number

If you specify user_names, then a full listing of information is displayed for each user you specify. This listing includes:

- User's home directory
- User's login shell
- User's ~/.plan file
- User's ~/.project file

in addition to the first list of information presented above.

The **finger** command can also be used to look up user information across a network. Be aware of a time delay for the information to be returned. On a local network it should respond within a few seconds, but it could be longer depending on the network traffic load and the remote system's load.

COMMAND FORMAT Following is the general format of the **finger** command.

```
finger [ -bfhilmpqsw ] user_name
finger [ -l ] "user@host"
```

BSD (Berkeley)

```
finger [ -l ] [ -m ] [ -p ] [ -s ] [ user_name ] [ last ] [ first ]
finger "user@host"
```

Options The following list describes the options and their arguments that may be used to control how **finger** functions.

-b	Suppress the user's home directory and login shell from a long format display.
-f	Suppress the default header from short format displays.
-h	Suppress the .project file from a long format display.
-i	Display "idle" output format, only the login name, terminal line, login time, and idles time are displayed.
-l	Force a long output format. All of the information listed in the two previous lists is displayed.
-m	Match arguments only against user names (field one of the /etc/passwd file). By default **finger** matches the arguments you provide against the username, first names or last names.
-p	Suppress the display of the .plan files.
-q	Display "quick" output format, only the login name, terminal line, and login time are displayed.
-s	Force a short output format. Only the first set of information listed above is displayed.
-w	Suppress the full user name from a short format display.

BSD (Berkeley)

Only the -l, -m, -p, and -s options are supported. They have the same meaning as the System V options.

Arguments The following list describes the arguments that may be passed to the **finger** command.

first	The first name of a user. **finger** searches the fifth field of the /etc/passwd file for a match.
last	The last name of the user. **finger** searches the fifth field of the /etc/passwd file for a match.
user_name	The name of the user you want information about.
user@host	Requests information about a user residing on the system named *host*.
	If no user is specified, information is returned for all current users logged into the system.

DIAGNOSTICS AND BUGS The **finger** program only displays the first line of a user's $HOME/.project file, even if multiple lines exist.

If **finger** seems slow, it may be caused by a large /etc/passwd file. To help speed up **finger** use the -m option.

RELATED COMMANDS Refer to the **who** command described in Module 159.

RELATED FILES The following files are accessed by **finger** for the required information.

/etc/utmp	The current who file
/etc/passwd	The system's password file
/var/adm/lastlog	An administrative file used to track the last login times of users
~/.plan	The user's plan. In the academic world this was the major; in business it is usually future projects
~/.project	The user's current project

APPLICATIONS

The **finger** command provides a quick way to retrieve information about a certain user or all users currently logged into the system. You can use **finger** to display information about a user you want to contact.

Remember, like any database, the information returned is only as useful as it is current. If you have a good system administrator that keeps the necessary files updated, then **finger** will prove to be a useful tool. If the data is outdated, **finger** will be a waste of time for certain information.

TYPICAL OPERATION

In this activity you use the **finger** command to display information about all current users. Begin at the shell prompt.

1. Type **finger** and press **Return**. This will display information about each user currently logged on to the system.

In this activity you use **finger** to display all the information about your user name. Then update your .project file and rerun finger to see the difference.

1. First type **finger mylogin** and press **Return**. Notice certain fields may be empty.
2. Type **cd** and press **Return** to change to your home directory.
3. Now type **vi .project** and press **Return** to edit your .project file.
   A. Type **i** to enter insert mode.
   B. Enter the desired text, only one line.
   C. Press **Escape** to exit insert mode.
   D. Type **ZZ** to save your file and exit vi.
4. Type **finger mylogin** and press **Return** to display the additional information about yourself.
5. Turn to Module 75 (SV), Module 160 (BSD) to continue the learning sequence.

Module 57
fmt

DESCRIPTION

The external **fmt** command performs simple right margin justification text formatting. A list of files are read in and the input is massaged to be as close as possible to 72 columns wide. Spacing at the beginning of input lines, blank lines, and spacing between words are left unmodified. If no files are provided as arguments, **fmt** reads from the standard input.

The **fmt** command was originally written to format messages before mailing, but it may be useful for other simple formatting tasks. It is useful for formatting sections of text printed by a shell script.

COMMAND FORMAT Following is the general format of the **fmt** command.

```
fmt [ -cs ] [ -w width ] [ file_list ]
```

BSD (Berkeley)

```
fmt [ filename_list ]
```

Options The following list describes the options and their arguments that may be used to control how **fmt** functions.

-c	Use crown margin mode. The indentation of the first two lines is preserved, with the rest of the paragraph aligned with the second line.
-s	Do not join short lines to form longer ones. Only split lines. This prevents short lines, such as examples of program code, from being combined.
-w *width*	Format output text up to *width* columns. The default width is 72 columns.

Arguments The following argument may be passed to the **fmt** command.

file_list	A list of files to format. The fmt command is commonly used without files being specified, that is it is often a filter.

EXAMPLE A popular use of **fmt** is from within the **vi** editor. Assume you have created the following file and you are on line one. The cursor is under the T of *This*.

```
This is a brief message to the Control Unit group.
The new hp is now on-line. To send prints to it, type lj filename.
If you have questions contact Sys. Adm. (x480).
The printer is located in the stock room.
```

Now type **!}** **fmt** and press Return. This command sends the paragraph to the **fmt** command as standard input. Then **vi** reads the standard output of the **fmt** command and replaces the paragraph with the formatted output. Your paragraph now resembles the following screen.

```
This is a brief message to the Control Unit group. The new hp is now
on-line. To send prints to it, type lj filename. If you have questions
contact Sys. Adm. (x480). The printer is located in the stock room.
```

Obviously **fmt** is very limited in what it can do, but it is very fast for simple text formatting.

RELATED COMMANDS Refer to the **fold** command described in Module 58.

RELATED FILES The **fmt** command reads input from a list of files or from the standard input. It writes to the standard output.

APPLICATIONS

You can use the **fmt** command to format messages before sending them to another user via **mail**. Another use is for formatting memos or comments in source code.

TYPICAL OPERATION

In this activity you use the **fmt** command to format the entire buffer of text while you are in **vi**. Begin at the shell prompt.

1. Type **vi fmted** and press **Return**. This will put you in **vi**.
2. Type **i** and enter two or three paragraphs of text, such as the following.

```
cj> vi fmted
i
The fmt command will format this buffer screen.
By using the vi filter interface it is sent out to the standard input of
a UNIX command and the standard output of the command is read back into
the vi buffer.
The contents of the buffer are overlayed with the new output
of the command.

The fmt command will reformat the jagged right end of the lines to be as
close as possible to 72 columns.
It basically only performs right margin justification.
```

3. Now type **1G** to move to the top of the buffer. The cursor is at the top left corner of the screen (buffer).
4. Type **!Gfmt** and press **Return** to send the contents of the buffer to the **fmt** command. The buffer will be redrawn with the newly formatted output. The output should resemble the following.

```
The fmt command will format this buffer screen. By using the vi filter
interface it is sent out to the standard input of a UNIX command and the
standard output of the command is read back into the buffer. The contents
of the buffer are overlayed with the new output of the command.

The fmt command will reformat the jagged right end of the lines to be as
close as possible to 72 columns. It basically only performs right margin
justification.
```

5. Type **:q!** and press **Return** to exit **vi** without saving the text.
6. Turn to Module 58 to continue the learning sequence.

Module 58
fold (BSD)

DESCRIPTION

The external **fold** command is a filter that folds lines longer than 80 characters into lines less than 80 characters. You can specify the width of the line instead of using the default of 80.

If tabs are present in the input data, you may want to send the data through **expand** first, then pipe it into **fold**.

COMMAND FORMAT Following is the general format of the **fold** command.

```
fold [ -wwidth ] [ file_list ]
fold [ -width ] [ file_list ]
```

BSD (Berkeley)

```
fold [ -width ] [ file_list ]
```

Options The following option and its argument may be used to control how **fold** functions.

-width
-wwidth — Specifies the width of the output line in characters. The default is 80. The *width* should always be a multiple of your tab length if tabs exist within the input data. Usually tabs are set every eight spaces.

Arguments The following argument may be passed to the **fold** command.

file_list — A list of files to be read and checked for lines longer than the specified or default width.

DIAGNOSTICS AND BUGS The **fold** command may interfere with underlining in the input text.

RELATED COMMANDS Refer to the **expand** command described in Module 46.

RELATED FILES The **fold** command reads a list of files or uses the standard input if no files are specified as arguments. The output is written to the standard output.

APPLICATIONS

You use **fold** to break long lines down into shorter more readable lines. This is a very useful function if you have a terminal that does not wrap its lines onto the next line. It may also be used for printers that do not know how to wrap lines longer than a certain width onto the next line. By using **fold** you can perform the wrapping function before the data reaches the output device.

Some terminals automatically wrap long output lines onto the next line. This looks good but may cause problems with commands that attempt to count displayed lines. If the line is longer than 80 characters, it takes up two lines of your terminal screen. Programs like **pr** format the output based on the number of physical lines read. Thus a line may scroll off the top of your screen or your printed output may not be aligned on the paper correctly, because the number of physical lines does not match the number of displayed lines.

The fold command may be used to correct this problem, because it breaks the long lines down into shorter physical lines, making other commands count the proper number of physical lines.

TYPICAL OPERATION

In this activity you use the **fold** command to wrap long lines onto the next output line. First you must create a file with lines longer than 80 characters. Begin at the shell prompt.

1. Create a file with two or three lines of text longer than 80 columns. Type **vi xwide** and press **Return**. You may need to **:set wm=0** to prevent **vi** from inserting automatic new-lines in your text.
2. Type **i** and enter the following text; press **Return** only where the <CR> notation appears.

```
cj> vi xwide
This is a line of text that is longer than 80 characters. It wraps around
to the next line on some terminals but not on others. Therefore we use
fold.<CR>
Another line just for filling up space. The fold command cuts these lines
off at 80 characters (or there about) and displays the remainder of the
line on the next output line.<Escape>
```

3. Type **ZZ** to save the contents and exit the editor.
4. Type **cat xwide** and press **Return**. If five lines of text appear, your terminal wraps long lines onto the next line automatically. If only two lines appear, you have a terminal that is not wrapping the lines.
5. Type **fold xwide** and press **Return.** Notice the lines are displayed from 1 to 80 and 81 to 132 on the next line.
6. Type **rm xwide** to remove the file created in this activity.
7. Turn to Module 105 to continue the learning sequence.

Module 59
getopts (ksh)

DESCRIPTION

The internal **getopts** command is used in shell scripts to parse the positional parameters. It checks for legal options as specified in the command rules. Refer to Module 23 describing Commands for a list of the rules. The **getopts** command supports rules 3 through 10. It replaces the previous command parser **getopt** which did not support all of the command rules.

COMMAND FORMAT Following is the general format of the **getopts** command.

```
getopts optstring name [ arg ... ]
getopts optstring -- name [ arg ... ]
```

Arguments The following arguments may be passed to the **getopts** command.

optstring Must contain valid options your shell recognizes. If a character is followed by a colon (:), an option argument is expected. The argument is stored in the variable OPTARG. For example, if you plan on using options i, o that has an argument, and v, then you would have to write your *getopts* command as follows:

```
getopts vio: OPTION
```

valid command lines would be:

```
cmd -iv -o oarg file
cmd -iv -o oarg -- file
cmd -i -v -o oarg file
cmd -v -o oarg -i file
cmd -o oarg -i -v file
```

Command line options must be preceded by a - or + sign.

name Each time **getopts** is invoked (called within the shell) the next option is placed in the shell variable *name*. For example, in the code

```
while getopts abc: OPTION
...
```

with a command line of

```
cmd -b -a -c myfile
```

the OPTION variable becomes b on the first iteration of the loop, a on the second, and c on the third.

If a "+" is attached to the option on the command line, a "+" is attached to the beginning of the value of the *name* variable.

arg If *arg*uments are specified, **getopts** will parse them instead of the positional parameters passed to the shell script at invocation time. For instance,

```
getopts abc: OPTION a c afile
```

causes a, c, and afile to be used as positional parameters 1, 2, and 3, respectively.

-- Delimits the end of all options and the beginning of all arguments if any exist.

Command Line The command line must conform to the standard rules specified in the Command Rules section of Module 23.

Command line options must be preceded by a - or + sign.

The first command line argument that is not preceded by a - or + sign is considered the start of all arguments and the end of all options.

FURTHER DISCUSSION The index (positional parameter number) of the next argument to be processed is placed in the shell variable OPTIND. For example, based on the following two commands:

```
cmd -a -b -c file
cmd -ab -c file
```

OPTIND is set to 2 after the -a option is processed, because the next option to be processed (b) is in the second positional parameter. In the second command OPTIND is set to 1 after the -a option is processed, because the next option to be processed (b) is in first positional parameter($1). OPTIND is initialized to one when the shell is first invoked.

If an option requires an argument (a colon was placed after it in the *optstring*), the argument is placed in the shell variable OPTARG.

Invalid Options and Optargs Depending on the *optstring* layout **getopts** handles invalid options in different ways.

If *optstring* begins with a : and an invalid option is encountered, **getopts** puts the bad option in the shell variable OPTARG and sets *name* to ?. If *optstring* begins with a : and an option argument is omitted, **getopts** sets *name* to : and sets OPTARG to the options encountered.

If *optstring* does not begin with a : and an invalid option is encountered, **getopts** sets *names* to ? and displays an error message.

The **getopts** command is invoked for each iteration of the while loop. Thus OPTION is set to the value of the next option on the command line. The **case** uses the value in OPTION to determine what function is to be performed.

The following example illustrates how you can use the **getopts** command in a shell script.

```
while getopts :abc: OPTION
 do
  case $OPTION in
   a) A_FLG="yes" ;;
   b) B_FLG="on"  ;;
   c) C_OPT=$OPTARG ;;
   :) echo "$0: $OPTARG missing argument!"
      exit 2 ;;
  \?) echo "$0: $OPTARG is an invalid option!"
      echo "$0 -ab -c optarg file_list"
      exit 2 ;;
  esac
 done
shift $OPTIND-1
```

DIAGNOSTICS AND BUGS If **getopts** encounters an option not specified in the *optstring*, it displays an error message on the standard error.

RELATED FILES The **getopts** command displays its output to the standard output.

APPLICATIONS

The **getopts** command provides a standard command parser for your shell scripts. If you learn to use it to parse the options for your shell scripts, you won't have to code for valid combinations of options. It checks for valid options then formats them into an easy-to-work-with string. If you are using the csh, skip to the next module.

TYPICAL OPERATION

In this activity you use the **getopts** command to parse the command line arguments and **echo** them back to your screen. Begin at the shell prompt.

1. Type in the following lines of shell code using the **vi** editor. Call the file *do_opts*. Type **vi do_opts** to enter the editor.

```
while getopts :abc: OPTION
 do
  case "$OPTION" in
   a) A_FLG="yes"    ; echo "A_FLG=$A_FLG" ;;
   b) B_FLG="on"     ; echo "B_FLG=$B_FLG" ;;
   c) C_OPT=$OPTARG ; echo "C_OPT=$C_OPT ;;
   :) echo "$0: $OPTARG option missing argument!"
      exit 2 ;;
   ?) echo "$0: $OPTARG is an invalid option!"
      echo "$0 -ab -c optarg file_list"
      exit 2 ;;
  esac
 done
shift $OPTIND-1
```

2. When you have finished, save the file and change the modes to 755 by typing **chmod 755 do_opts**.

3. Now test the new command by typing **do_opts -a -c afile** and pressing **Return.** Your output should resemble the following.

```
cj> do_opts -a -c afile
A_FLG=yes
C_OPT=afile
```

4. Type **do_opts -b -x afile** and press **Return** to cause an error in the options. Your screen should resemble the following display.

```
cj> do_opts -b -x afile
B_FLG=no
do_opts: -x is an invalid option!
do_opts -ab -c optarg file_list
```

5. Type **do_opts -b -a -c** and press **Return** to cause an error in the option argument required for option c. Your screen should resemble the following display.

```
cj> do_opts -b -a -c
B_FLG=b
A_FLG=a
do_opts: -c option missing argument!
```

6. Turn to Module 6 to continue the learning sequence.

Module 60
grep

DESCRIPTION

The external **grep** command is used to search ASCII text files for a given pattern. All lines containing the pattern are written to the standard output. The pattern may be a word or string of characters. Regular expressions may be used inside the pattern to specify certain sequences of characters. If **grep** searches more than one file, each displayed line is preceded by the filename. If no filenames are given, **grep** reads from the standard input. Thus you can use **grep** in a pipe command.

The name **grep** is a combination of editor command characters. It is from the editor command :g/RE/p, which translates to global Regular Expression print. Inside the editor this command would search the entire file for all lines matching the RE *pattern*. The -v option of **grep** is from the editor command :v/RE/p, which searches the entire file for all lines except the ones containing the RE *pattern*.

NOTE

The **grep** command performs line oriented searches. Therefore, a phrase that spans more than one line is not matched. When deciding what string to use for the search you should try to keep it as short as possible yet as unique as possible.

COMMAND FORMAT Following is the general format of the **grep** command.

```
grep [ -bchilnsv ] pattern file_list
```

BSD (Berkeley)

```
grep [ -bcilnsvw ] pattern file_list
```

Options The following options may be used to control how **grep** functions.

-b Displays the block number in which the *pattern* was found before the line that contains the matching *pattern*.

-c Displays only a total count of matching lines for each file processed.

-h Suppresses the displaying of filenames which precede lines that match the specified patterns when multiple files are searched.

-i Ignores the difference between uppercase and lowercase characters during comparisons.

-l Displays only the names of the files containing the specified *pattern*. The lines containing the *patterns* are not displayed.

-n	Displays the line number before each line containing the *pattern*.
-s	Suppresses the displaying of diagnostic error messages for nonexistent and nonreadable files.
-v	Displays only the lines that do not match the *pattern*. The v command in the **ex** editor performs the same type of function. It is an exception search. Search for every line except the ones containing the given *pattern*.

BSD (Berkeley)

-e *-string*	This allows you to specify a string that begins with a dash. Normally any argument beginning with a dash is interpreted as an option not a string or argument.
-w	The expression is searched for as a word. The RE metacharacters \< and \> are assumed to surround the expression.

Arguments The following list describes the arguments that may be passed to the **grep** command.

pattern	Any combination of characters, numbers, and regular expression patterns. **grep** searches for strings of text that match the *pattern* expression.
file_list	One or more files to search for the given *pattern*

SEARCH PATTERNS

(REGULAR EXPRESSIONS) The following table contains each regular expression and the task that it performs when used inside a *pattern*. A regular expression consists of regular alphanumeric characters matching themseleves and special characters matching certain patterns of text. Regular Expressions are often referred to as REs in UNIX terminology. Thus we use the RE notation for uniformity and briefness.

Alphanumeric RE	*Description*
c	Matches the character *c*.
string	Matches the specified string of characters.

Metacharacters Metacharacters are the special characters used in regular expression patterns that have special meanings. Metacharacters are often referred to as *special* or *magic* characters or wild cards.

Special RE	*Description*
\	Escapes the meaning of a metacharacter.
^	Matches the beginning of the line.
$	Matches the end of the line.
.	Matches any single character.

[*class*]	A character class. Matches any one character in the class.
[*c1-c2*]	Match any one of the ASCII characters in the range defined within the brackets.
[^*class*]	Does NOT match any of the ASCII characters listed within the brackets. Ranges may be specified.
*	Matches zero or more occurrences of the preceding regular expression.
\(RE\)	Groups the regular expression RE for later reference; each RE is referenced by a corresponding number (see \n). The first group is referenced by \1, the second group by \2, and so on. You can have a maximum of nine groups in one regular expression pattern.
\n	Matches the *n*th grouped \(RE\) within the same regular expression.
RE\{*m*\}	Matches exactly *m* occurrences of the preceding one-character RE.
RE\{*m*,\}	Matches *m* or more occurrences of the preceding one-character RE.
RE\{*m*,n\}	Matches *m* through *n* occurrences of the preceding one-character RE.

To have a metacharacter interpreted as a normal character, proceed it with a backslash (\).

DIAGNOSTICS AND BUGS The input lines read by **grep** are limited to the system's BUFSIZ definition. If a line is longer than BUFSIZ characters, the line is truncated before the comparison is made. The BUFSIZ used by your system can usually be found in the /usr/include/stdio.h file.

RELATED COMMANDS Refer to the **nawk**, **ed**, **ex**, **egrep**, **fgrep**, **sed**, and **ksh** commands described in modules 6, 39, 43, 40, 52, 117, and 71.

RELATED FILES The **grep** command can read from the standard input or a specified list of files. It writes to the standard output.

RETURN CODES The return code is set to 0 if any *patterns* where found; if no *patterns* where found it is 1. If syntax errors are present in the *pattern* or if files are not accessible, a return code of 2 is returned, even if *patterns* where found.

APPLICATIONS

You can use **grep** to search text for a specific word or string of characters. The text may be contained in a file or be from the standard input. The various uses of grep include finding lines or files that contain a specific *pattern*, counting the number of lines that contain the *pattern* in a file, showing the line numbers of lines that contain the *pattern*, and showing the block that contains the *pattern*. These uses may be inverted to function on all lines that do not contain the specified *pattern* also.

The uses of **grep** are far reaching, from just searching a file for lines that contain a *pattern* to using **grep** inside shell scripts to verify a *pattern* exists. It is one of those commands that doesn't seem useful until you use it a few times, then it becomes one of your more commonly used commands.

TYPICAL OPERATION

In this activity you use the **grep** command to display the BUFSIZ for your system, then to locate your login in the passwd file. Various *pattern* examples are also used to show you how to use the regular expressions. Begin at the shell prompt.

1. Type **grep BUFSIZ /usr/include/stdio.h** and press **Return**. This returns the maximum number of characters a line can contain that **grep** can search for in a given line.
2. Type **grep mylogin /etc/passwd** and press **Return**. Notice your account entry in the /etc/passwd file is displayed.
3. Type **grep -v mylogin /etc/passwd** and press **Return**. Notice that all lines except the one containing your account are displayed on your screen.
4. Type **who | grep tty07** and press **Return**. You may have to change the tty07 pattern to a different pattern, but this command returns who is logged in on the terminal device tty07.
5. Turn to Module 40 to continue the learning sequence.

Module 61
groups

DESCRIPTION

The external **groups** command displays the groups to which you or a user belongs. All users belong to a group. The fourth colon-separated field of the /etc/passwd file defines the primary group to which you belong. You may also belong to other groups by being listed beside a group in the /etc/group file. You have group permissions for files or directories owned by one of the groups to which you are a member.

COMMAND FORMAT Following is the general format of the **groups** command.

```
groups [ user ... ]
```

Arguments The following argument may be passed to the **groups** command.

user You can specify a list of users whom you want to know to which groups they belong.

RELATED FILES The **groups** command reads your user name and group ID from the /etc/passwd file, then searches the /etc/group file for any additional groups to which you belong.

APPLICATIONS

The **groups** command is used to display all of the groups to which you belong. You may find it helpful to belong to several groups so you have certain access permissions for various directories and files.

TYPICAL OPERATION

In this activity you use the **groups** command to display your current group membership list. Begin at the shell prompt.

1. Type **groups** and press **Return** to display all of the groups to which you are a member.
2. Turn to Module 161 to continue the learning sequence.

Module 62
hash (ksh)

DESCRIPTION

The aliased **hash** command forces a tracked alias for a specific command. It is an alias for the **alias -t** command. A tracked alias is the full pathname for a command. This reduces the time it takes the shell to search for the command and execute it. Since the full path is known to the shell through the alias name, the shell does not have to search each directory listed in the PATH variable, thus reducing the time it takes to invoke a command.

COMMAND FORMAT Following is the general format of the **hash** command.

```
hash command
```

Arguments The following argument may be passed to the **hash** command.

command The name of the command you wish to have tracked. For example,

```
hash ls
```

forces the **ls** command to be a tracked alias.

If no argument is provided, then all tracked aliases are displayed.

Automatic hashing You can use the **set** command to have all commands you enter on the command line become tracked aliases. The command

```
set -o trackall
```

or

```
set -h
```

causes automatically tracked aliasing.

RELATED COMMANDS Refer to the **alias** command described in Module 4.

APPLICATIONS

The **hash** command is used to create tracked aliases for commands. Tracked aliases provide the shell with the entire pathname for a command. This allows the shell to locate the command faster and thus less overhead is needed to execute a command.

TYPICAL OPERATION

In this activity you use the **hash** command to create a tracked alias for the **vi** command and then display all tracked aliases. If you're using the csh, skip to the next module. Begin at the shell prompt.

1. Type **hash vi** and press **Return**. This creates a tracked alias for **vi**.
2. Type **hash** and press **Return** to display all tracked aliases.

```
cj> hash
vi=/usr/bin/vi
```

3. Turn to Module 68 (SV), Module 9 (BSD) to continue the learning sequence.

Module 63
head

DESCRIPTION

The external **head** command displays the first few lines of a file. It displays however many lines you specify as an option. If you do not specify the number of lines to display, **head** defaults to ten. Its cousin is the **tail** command.

COMMAND FORMAT The format of the **head** command follows.

```
head [ -n ] file
```

Options The following describes the option used to control how **head** functions.

-n The number of lines to display at the beginning of each file.

Arguments The following describes the argument that may be passed to the **head** command.

file_list One or more files to be displayed by head. If the *-n* is specified, it applies to all files in the *file_list*.

FURTHER DISCUSSION If more than one file is specified on the command line, **head** precedes each file with a line like:

```
===> filename <===
```

If only one file is given on the command line, **head** does not inform you of the *filename* being displayed.

You may choose to use **head** to display the entire contents of multiple short files so each file will be preceded by its filename. The following command provides an example:

```
cj> head -999 file1 file2 file3
```

RELATED COMMANDS Refer to the **cat**, **pg**, **pr**, **sed**, and **tail** commands described in modules 13, 103, 105, 117, and 131, respectively.

RELATED FILES The **head** command can read from a file or from the standard input. The **head** command writes to the standard output. It can therefore be used at the beginning of a pipe. Its output can be redirected to a desired file.

APPLICATIONS

The **head** command is useful for verifying what type of data is in a file. It can be used to view the first few lines of an ASCII file to visually decide what data is contained in the file. You might consider **head** as a special version of the **cat** command, allowing you to view the beginning portion of the file.

TYPICAL OPERATION

In this activity you use the **head** command to display the first 15 lines of the /etc/passwd file. Begin at the shell prompt.

1. Type **head -15 /etc/passwd** and press **Return**. Your screen scrolls up as the first 15 lines of the password file are displayed.
2. Turn to Module 131 to continue the learning sequence.

Module 64
history (csh, ksh)

DESCRIPTION

The aliased **history** command displays your command history file. Each command you enter can be saved in a command history file. You can later access this history file to reexecute a previous command. It is common practice to alias **h** for **history**. The Korn shell alias and csh alias lines follow:

```
alias h=history      # Korn shell
alias h history      # C shell
```

You want to place these commands in your .profile or .kshrc if you use the Korn shell (ksh) or in your .login or .cshrc if you use the C shell (csh).

When you log out of the system your history file is removed unless you inform the shell to save it.

COMMAND FORMAT Following is the general format of the **history** command.

```
history [ -nr ] [ -b [ -e ] ]
history [ -nr ] [ b [ e ] ]
```

C Shell

```
history [ n ]
history -r [ n ]
history -h [ n ]
```

Options The following options may be used to control how the **history** command functions.

-b Beginning point. The number of lines to go back from the last command to start listing out the history file. For example,

```
history -25
```

lists the last 25 commands saved in the history file.

-e Ending point. The number of lines to go back from the last command to end listing out the history file. For example,

```
history -25 -5
```

lists out the last commands starting 25 commands ago down to 5 commands ago.

-n Suppress the leading command numbers from the history listing.

-r Reverse the order of history file for displaying. The latest commands issued are displayed first.

C Shell

-h Same as **ksh** -n option. This may prove useful if you wish to create a file you can **source** or **.** (dot) into your local shell.

-r Same as **ksh** -r option

Arguments The following list describes the arguments that may be passed to the **history** command.

b List commands starting at line *b* in the history file. For instance,

```
history 15
```

displays the commands beginning at line 15 to the end of the history file.

e List commands up to line *e* in the history file. For example,

```
history 20 30
```

displays lines 20 through 30 of the history file.

C Shell

n Displays the *n* most recent commands (events) stored in the history file.

Variables There are two variables that control the location and size of the history file. The HISTFILE variable informs the shell where to locate the history file. For example,

```
HISTFILE=$HOME/.history
```

places the history file in your HOME directory in the .history file. If you do not specify path in the HISTFILE variable, the default for the location is $HOME/.sh_history.

The HISTSIZE variable is used to set the number of commands stored in the history file. If you do not set the HISTSIZE variable, it defaults to 128. Therefore, the history file will contain the last 128 commands you have entered.

C Shell

The C shell uses different variables than the Korn shell. To control the size of the file, you use the **history** variable. For example,

```
set history=100
```

would cause the shell to save the last 100 commands in the history file.

The **savehist** variable is available in the csh to save *n* commands in your history file when you log out. The next time you log back into the system those *n* commands are still available in your history file. If you do not set **savehist** to a number of history commands to save, your history file is recreated each time you log in.

You can set the **savehist** variable by placing the **set savehist=50** command in your .login file. This will save the last 50 commands in your history file when you log out of the system. The next time you log in, the last 50 commands will be available in your history file.

Another variable may be set to change the history access characters. By default you type ! or ^ to access a previous command residing in the history file. You can change these characters to alternate characters by setting the **histchar** variable. For example,

```
set histchar=.,
```

changes your history substitute characters to a period (.) and a comma (,). Now you type a .34 to execute the 34th command in your history file. To perform quick substitution on the last command executed, use the , notation instead of the caret (^). For example, type ,-xa,-la, to change xa to la in your last command.

If the **verbose** variable is set, each command is displayed after history substitution is performed. You set this variable using the -v option of the **csh**.

RELATED COMMANDS The **read** command will place its input in the history file if you specify the -s option (**ksh** only). Refer to Module 112 for further discussion on the **read** command.

RELATED FILES The history file is located based on the HISTFILE variable (**ksh** only). Usually, you set this variable to your home directory. For example, type

```
HISTFILE=~/.history
```

to place your history file in your home directory in a hidden file. It is common practice to place this variable assignment in your .profile file. The default history file is $HOME/.sh_history.

APPLICATIONS

The **history** command is used to list previous commands so you can reuse them using the **r** command (**fc -e -** in **ksh**) or the **!** notation (in **csh**). If you forget what number a command is assigned, you can use the history command to list the commands so you can locate the appropriate number.

TYPICAL OPERATION

In this activity you use the **history** command to list the history file. Begin at the shell prompt.

1. Type **history -23** and press **Return**. Notice the last 23 commands of your history file are displayed on your screen.

C Shell

If you use the **csh**, type **history 23** and press **Return** to display the last 23 commands.

---

2. Turn to Module 50 to continue the learning sequence.

Module 65
hostname (BSD)

DESCRIPTION

The external **hostname** command sets or displays the name of the current host system. It is a network related command. System V based operating systems do not support **hostname** unless networking software has been installed. All BSD based operating systems support **hostname** as a standard command. It is a simple substitute for the **uname** command.

Only the super-user can change the hostname of the computer. By specifying a new hostname as an argument to **hostname** the super-user can change the system's hostname. The hostname is usually set at boot time in the /etc/rc.local on BSD systems. System V systems usually use the **uname -n** commands to set the hostname during system boot in the /etc/rc or /etc/rc2.d/SXX file.

COMMAND FORMAT Following is the general format of the **hostname** command.

```
hostname [ name_of_host ]
```

Arguments Only one argument may be passed to the **hostname** command.

name_of_host The name of your local host system. The computer you are currently logged into.

RELATED COMMANDS Refer to the **uname** command described in Module 148.

APPLICATIONS

You will find the **hostname** command useful if you use systems connected to one another via a LAN. The standard UNIX LAN is Ethernet using the TCP/IP protocol. When you log in from one system to the next you may find yourself forgetting which system you are currently logged into. By using **hostname** you can find out the system's name.

Many users set their shell prompt to the output of hostname. For example,

```
PS1="`hostname`> "
```

TYPICAL OPERATION

In this activity you use the **hostname** command to display the name of the system you are currently using. Begin at the shell prompt.

1. Type **hostname** and press **Return**. Notice the name of the system is returned.
2. Turn to Module 144 to continue the learning sequence.

Module 66
id (SV)

DESCRIPTION

The **id** command displays your user ID, user name, group ID, and group name. Your user ID is the third field of the password file. The system uses this user ID to identify the files you own and other important information about processes you are executing. It converts (maps) your user ID to your user name and vice versa. The group ID works the same as the user ID except it is used for group level identification.

If the effective and real IDs are different, both are displayed. The effective IDs are what the shell environment presently considers your user ID and group ID. The real IDs are the user ID and group ID that you obtained when you originally logged in to the system.

COMMAND FORMAT The general format for the **id** command is as follows.

```
id [ -a ]
```

Options The following option may be used to control how the **jobs** command functions.

-a Displays all the groups to which the invoking process belongs. If the real and effective ids are different, both are displayed.

RELATED COMMANDS Refer to the **who** command described in Module 159 and the **logname** command described in Module 78.

APPLICATION

The **id** command is used to display your user ID and group ID. The user name and group name are also displayed. The utility is useful for checking a user's ID and access privileges from a shell script that requires a certain level of security or logging information. It is also useful to check who has left themselves logged on at an unattended terminal.

TYPICAL OPERATION

In this activity you use the **id** command to display your user ID, group ID, user name, and group name.

1. Type **id** and press **Return**. Notice the user ID is displayed, followed by the user name enclosed in parentheses. The group ID and group name have the same format.

```
cj> id
uid=101(mylogin) gid=40(tech)
```

2. Turn to Module 78 to continue the learning sequence.

Module 67
INPUT/OUTPUT AND REDIRECTION

DESCRIPTION

Many UNIX commands read from the standard input, write to the standard output, and report errors to the standard error. When you run a command it opens the standard input, output, and error automatically. The standard input, by default, is your keyboard; the standard output and error is your terminal screen. Thus a command reads from your keyboard if it reads the standard input, and writes to your terminal screen if it writes to the standard output or error.

These three standard I/O files (devices) are assigned file descriptors. File descriptors are numbers that relate to an open file. If a command opens another file for reading or writing, the next available file descriptor is assigned to the file. The following table describes each I/O type and the related default file descriptor.

STANDARD INPUT/OUTPUT/ERROR

Conventional Names	*Default Device Input or Output is Connected to*	*File Descriptor (Used by the Shell)*
Standard Input	Received from terminal keyboard (or from a pipe or file)	0
Standard Output	Sent to the terminal screen (or to a pipe or file)	1
Standard Error	Sent to the terminal screen (or to a pipe or file)	2

REDIRECTION The shell allows redirection of the standard input, output, and error of a command. *Redirection* of I/O is the capability to change the source of the input and the destination of the output. This is done by using the greater than and less than signs (<,>) to guide the output. The redirection symbols(<,>) are not passed to the command as arguments; they are interpreted by the shell. The symbols may appear anywhere on a simple-command line or may precede or follow a command. The pipe command (|) also redirects input and output between programs. It is not considered a redirection symbol because it is used between commands. Redirection symbols affect the source input or the destination output of a command.

CAUTION

You should be careful when redirecting I/O. It is possible to overwrite files or lock up your terminal. If you overwrite a file, your system administrator may be able to recover it from the last backup. If you lock up your terminal, contact your system administrator for assistance.

NOTE

A single digit may precede any of the redirection symbols to change the default file descriptor that is affected.

GENERAL FORMAT The general formats of the redirection symbols follow.

<name

The standard input is read from file *name* instead of the terminal keyboard. For example,

```
sort < unsorted.data
```

causes the shell to reconnect standard input from your keyboard to the file *unsorted.data*. When the **sort** command opens the standard input and begins to read, it will read from the *unsorted.data* file, not your keyboard. After **sort** completes, the shell reconnects the standard input to your keyboard.

>name

The standard output is sent to the file *name* instead of the terminal screen. If the file *name* does not exist, then it is created; otherwise, it is overwritten. For instance,

```
sort unsorted.data > sorted.data
```

would sort the *unsorted.data* file and redirect it to the *sorted.data* file.

>|

Forces a write to a file that exists even though you have the *noclobber* option set to prevent such action.

>>name

The standard output is appended to the file *name*. If the file does not exist, then it is created. For example,

```
sort more.data >> unsorted.data
```

reads the data in the *more.data* file and appends it to the end of the *unsorted.data* file.

<<[-]*name*

This is referred to as a *here document*. It allows you to enter multiple lines of input as a block of standard input. The text is read from the standard input until a line containing only *name* is read. If any character in the first *name* is quoted, then no variable, command, or tilde substitutions are performed inside the here document input. If a - (hyphen) is present, all leading tabs are removed from the input text. The exclamation mark (!) is often used as *name*. For example,

```
tr '[a-z]' '[A-Z]' <<!
this text will be changed to uppercase
by the tr command.
The next line ends the input!
!
THIS TEXT WILL BE CHANGED TO UPPERCASE
BY THE TR COMMAND.
THE NEXT LINE ENDS THE INPUT!
```

<&fd

Read the standard input from the file associated with file descriptor *fd*. Same as the 0<*&fd* notation.

>&fd

Write the standard output to the file associated with file descriptor *fd*. Same as the 1>*&fd* notation. For example,

```
echo Error occurred after command on line $LINENO >&2
```

might be used in a shell script to write the standard output to the standard error.

nfd>&fd Create a new file descriptor nfd which is a duplicate of file descriptor *fd*. For example,

```
batchjob > batch.out 2>&1
```

sends the standard output of the **batchjob** program to batch.out and duplicates file descriptor 2 (the standard error) to be the same file as associated with file descriptor 1 (the standard output). The result is the standard output and standard error are written to the *batch.out* file.

<&- Close the standard input. Same as the 0<&- notation. For example,

```
exec <&-
```

closes the standard input. Not an intelligent move from your keyboard.

>&- Close the standard output. Same as the 1>&- notation. For instance,

```
2>&-
```

closes the standard error. Performs the same function as using 2> /dev/null. The standard error is not displayed. Again not always intelligent from your keyboard, but often useful in shell script programming.

<&p The standard input is connected to the background pipe. The pipe is established by a co-process when you end a command with &|.

exec < name Dynamically redirect input. Redirect standard input from the file *name*. The standard input for all subsequent commands is read from file *name*.

exec > name Dynamically redirect output. Redirect standard output to write to file *name*. The standard output of all subsequent commands will be redirected to file *name*.

C Shell

The C shell uses the same basic notation for I/O redirection as the Bourne and Korn shell. It does, however, support a few different notations. They are as follows.

> *name* Writes standard output to the named file. Checks the noclobber option; if it is set, an existing file is not overwritten. If it is not set, an existing file is overwritten.

>! *name* Writes standard output to the named file. Overwrites an existing file without checking the noclobber option.

>& *name* Reroutes the diagnostic output (standard error) to the standard output, so both are written to the named file.

>&! A combination of the previous forms. Overrides noclobber and places standard error with standard output.

|& Passes diagnostic output to the pipe along with standard output.

NOTE

The shell *keyword* commands can be redirected. This allows compound commands to be redirected to different output destinations.

Terminal Input and Output The shell can access a device name that always refers to your terminal screen and keyboard, /dev/tty. You can use /dev/tty to explicitly send the output of a command to the terminal screen, even if the output of the shell script has been redirected to a file or pipe. You can also use /dev/tty to explicitly read from the terminal keyboard. The formats for using /dev/tty follow:

```
echo "Press Return to Continue\c" > /dev/tty

read NAME < /dev/tty
```

Garbage Can Output The shell can access a device name that dispenses of all output. Referred to as the "garbage can," /dev/null can be used to redirect unwanted output. The format for using /dev/null follows:

```
if grep dog /etc/passwd > /dev/null
then  echo Someone has a login name of dog!
fi
```

Redirecting Output in Loops The standard I/O can be redirected in multiple ways within a shell or shell script. One of the best ways to show the flexibility of redirection is in a shell loop. The following example provides a brief view of how you can redirect I/O to and from different destinations within one loop.

```
cat somefile | \
while read LINE
 do
  read ANS < /dev/tty
  echo $ANS:$LINE > /dev/tty
  echo $ANS:$LINE
 done > someoutput
```

This example pipes the data in *somefile* to the standard input of the *while* loop. The **read LINE** command reads one line of *somefile* for each iteration of the while loop. The **read ANS** command reads a line of input from your keyboard. The /dev/tty device is a logical channel that is connected to your keyboard (and your terminal). The output of the first echo is sent to your screen. The output of the entire loop (the second **echo** command) is sent to the file *someoutput*.

If you did not use the "< /dev/tty" on the read command, the next line of the input file would be read into the ANS variable. And if the "> /dev/tty" notation was not used on the echo command, the output would go to the *someoutput* file.

APPLICATIONS

The primary application of Input/Output is for communicating with the computer system. By using the shells **read**, **select**, **echo**, and other commands, I/O can be presented in a formatted output and read in a formatted input. This allows for more user friendly menu building, data entry, or report generation.

Redirection is used to control where the input is read and where the output is sent. It can be used to store screen output to files. Input can be read from files, allowing you to enter all the necessary input in a file, then redirected to the desired command or file.

TYPICAL OPERATION

In this activity you use the redirection symbols to send output to a file and then have the shell redirect the standard input into a program. Begin at the shell prompt.

1. Type **ps -ef > psout** and press **Return**. Notice no output is returned to your screen. The output of ps was redirected to the *psout* file.

BSD (Berkeley)

On BSD systems type **ps -aux > psout** and press **Return**.

---

2. Type **cat < psout** and press **Return**. The file is displayed on your screen. The cat command is reading from the standard input. The shell opened the *psout* file and placed the data on the standard input, allowing cat to read the data. The result is the same as **cat psout**.
3. Type **notacommand** and press **Return**. You should get an error displayed.
4. Now type **notacommand > stdout 2> stderr** and press **Return**. Now type **cat stdout** and press **Return**. Nothing is displayed! Type **cat stderr** and press **Return**, notice the error is displayed. The shell sends its errors to the standard error not the standard output.

C Shell

Now type **notacommand >& stderr** and press **Return**. Now type **cat stderr** and press **Return**, notice the error is displayed. Also, note the csh redirects both the stdout and the stderr to the same file. This is one of the weaknesses of the csh output redirection function.

---

5. Type **rm psout stdout stderr** and press **Return** to remove the "psout" file.
6. Turn to Module 99 to continue the learning sequence.

Module 68
jobs (csh, ksh)

DESCRIPTION

The internal **jobs** command displays all processes in background or in a suspended state. Jobs are processes you have started in foreground, background, or have suspended. There are three states of jobs:

- foreground — A process whose I/O is attached to your terminal. If a process is executing in foreground, you have to wait until it completes to execute the next command.
- background — A process detached from your terminal. The process is executing while you continue to work at your terminal.
- suspended — A process that is not executing. It is waiting to be moved to foreground or background to continue execution.

If you end a command line with an ampersand (&) the process is immediately placed in background for processing.

If you start a command in foreground, you can suspend it by typing Ctrl-Z. You can place it in background with the **bg** command. The **fg** command allows you to reattach to the process in the foreground. A suspended job does not execute. It remains at the same instruction where you suspended it.

COMMAND FORMAT Following is the general format of the **jobs** command.

```
jobs [ -lnp ]
```

BSD (Berkeley)

```
jobs [ -l ]
```

Options The following options may be used to control how **jobs** functions.

-l	Lists the process IDs in addition to the normal output.
-n	Displays jobs that have stopped or exited since last notified.
-p	Displays only the process IDs. Not implemented on some systems.

BSD (Berkeley)

-l	Same as SV

OUTPUT The output of the **jobs** command provides you with the information you need to control jobs. We'll use the following output to explain the information provided by jobs.

```
cj> jobs -l
+[4]  5931   Stopped          vi myfile
-[3]  5917   Running          du -s / > /tmp/du.all
 [2]  5898   Done(0)          ls -R | lp
```

The first column contains the job number enclosed in square brackets. The plus (+) signifies the current job. The current job is the last job placed in background or stopped. The minus (–) signifies the previous job.

The second column contains the process ID (PID). Some systems display this as a default; others require the -l option.

The third column contains the current state of the job. The possible states are:

State	*Description*
Done(*n*)	The job has completed with return code *n*.
Running	The job is executing (processing) in background.
Stopped	The job has been suspended from execution.
Terminated	The job has been aborted via a signal.

Column four contains the name of the command you entered.

C Shell

The C shell provides an internal command to control notification of changes in job states. The command is **notify** ***%job.*** By entering this command the shell will notify you when the status of the *job* changes. Normally, the shell notifies you of changes in jobs before presenting the shell prompt.

If the *notify* shell variable is set, the **notify** ***%job*** command is automatically performed for each job you start.

Korn Shell

The **ksh** uses the -m option to control the job control feature. This is the "monitor" option. If the -m option is set, you can perform job control. If job control is enabled, you will be notified after each job control related command is performed.

RELATED COMMANDS Refer to the **bg, fg, kill, stop** and **suspend** commands described in modules 9, 51, 70, 125, and 129.

APPLICATIONS

You will need to use the **jobs** command to list any processes you have suspended or placed in the background. When a background job completes, the shell notifies you between commands. If you try to log out while you have suspended jobs, the shell warns you and does not log you out.

TYPICAL OPERATION

In this activity you use the **jobs** command to display all suspended and background jobs. Begin at the shell prompt.

1. Type **du -s / > /dev/null &** and press **Return**. This places a fairly long running job in background.
2. Type **vi x** and press **Return**.
3. Press **Ctrl-Z** to suspend the **vi** session.
4. Now type **jobs** to display the list of background and suspended jobs.

```
cj> jobs
-[1] 1423     Running        du -s / > /dev/null
+[2] 1428     Stopped        vi x
```

5. To kill the background job type **kill %-** and press **Return**. Notice you are notified after a couple of seconds that the process was terminated.
6. Now type **fg** to re-enter **vi**. The suspended **vi** process is brought back into interactive mode.
7. Type **:q!** to exit **vi**.
8. Turn to Module 129 to continue the learning sequence.

Module 69
join

DESCRIPTION

The external **join** command performs the relational database join function. It joins two files together based on a key field (column) that exists in both files. The files must be sorted on the key field. If the contents in the key field of both files match, the two lines are merged. The merged line is written to the standard output.

COMMAND FORMAT Following is the general format of the **join** command.

```
join [ -an ] [ -e str ] [ -jn m ] [ -o list ] [ -tc ] file1 file2
```

Options The following list describes the options and their arguments that may be used to control how **join** functions.

-a*n* If a field in file *n* does not have a match in the other file, the line is written to file *n*. For example, if you use

```
join -a1 file1 file2
```

and file1 has a line in it with a key field that is not matched in file2, then that line is written to the file named 1.

-e *str* Replaces empty output fields with the *str* string. The -o option causes empty fields in certain conditions.

-j*n m* The join is performed using the *m*th field of file *n*. If you do not specify *n*, the *m*th field is used from both files. The fields are numbered beginning with 1.

-o *list* Specifies the fields to be displayed on output after the join has been performed. The *list* is comprised of *n.m* descriptions that define which fields from each file are to be displayed. The *n* refers to the file name and the *m* refers to the field number. Multiple *n.m* descriptions may be used in the *list*. They must be space separated. An example is,

```
join -j1 3 -j2 4 -o 1.1 1.3 2.1 1.5 file1 file2
```

which uses field 3 of file1 and field 4 of file2 to join the files. The output is field 1 of file1, field 3 of file1, field 1 of file2, and field 5 of file1.

-t*c* Character *c* is used as the input and output field separator. For example,

```
join -t: file1 file2
```

informs **join** to use a : (colon) field separator in place of a tab.

The default input field separators are blanks (spaces, tabs, and new-lines). If the default separators are being used, multiple occurrences count as only one field

separator. Leading separators are ignored. The default output separator is a space.

Arguments The following list describes the arguments that may be passed to the **join** command.

- A hyphen may be used in place of file1 to cause **join** to read the standard input as *file1*.

file1 The first file to be joined to the second file. The file must be sorted in ASCII collating sequence on the fields that are used to join the file to *file2*.

file2 The second file to be joined to the first file. The file must be sorted in ASCII collating sequence on the fields that are used to join the file to *file1*.

JOINING FILES The joining of files is a relational database function. It involves the combining of two files (tables) that share a common field (key). Each file must be sorted on this field. The **join** command reads the first line of the first file, then reads lines from the second file until the contents in the key field of the second file match or exceed the contents in the key field of the first file. If a match occurs, the line from the first file and the line from the second file are combined and displayed on the output. If no match occurs, **join** begins the process again except reading is done from the first file. This process continues until one or both files have been read to the end and no more lines in the other file can match the last line of the file.

An example of a ***join*** command might help clarify how the join function works. Let's assume we have the following files.

file1:

01123	HP 9000/825
02213	Sun 3/110
03321	Sun 4/110
08412	HP 9000/835

file2:

01123	Hewlett Packard
02213	Sun Microsystems
03321	Sun Microsystems
08412	Hewlett Packard

The following command should produce the combined output as shown.

```
$ join -t"<Tab>" file1 file2
```

01123	HP 9000/825	Hewlett Packard
02213	Sun3/110	Sun Microsystems
03321	Sun4/110	Sun Microsystems
08412	HP 9000/835	Hewlett Packard

DIAGNOSTICS AND BUGS If fields are separated by spaces and tabs you need to use the **sort -b** command for sorting the file. If you use the -t option you need to use the **sort** command without the -b option.

Unfortunately, the **comm**, **join**, **sort**, and **uniq** commands have different format conventions for referencing data.

Numeric filenames may conflict with the -o option and the field descriptions that relate to it. One solution is to **mv** your files to temporary names beginning with a nonnumeric character. Then after the **join**, move the files back to their original names.

RELATED COMMANDS Refer to the **nawk**, **comm**, **cut**, **grep**, **paste**, **sort**, and **uniq** commands described in modules 6, 22, 29, 60, 101, 122, and 149.

APPLICATIONS

You use the **join** command to join related files together based on a key field. The output is a subset of the fields and lines from both files. It is useful in combining two related tables (files) together to create one larger table.

TYPICAL OPERATION

In this activity you use the **join** command to join the /etc/passwd and /etc/group file together to show the user name, user ID, group name, group ID, group name, and home directory. First we must insure that both files are sorted in ASCII collating sequence. Begin at the shell prompt.

1. Type **sort -t: +3.0n -4.0n +2.0n -3.0n /etc/passwd > mypasswd** and press **Return** to sort the passwd file on the group ID and user ID fields.
2. Type **sort -t: +3.0n -4.0n /etc/group > mygroup** and press **Return** to sort the group file on the group ID field.
3. Type **join -t: -j1 4 -j2 3 -o 1.1 1.3 2.1 2.3 1.6 mypasswd mygroup** and press **Return**.

```
cj> join -t: -j1 4 -j2 3 -o 1.1 1.3 2.1 2.3 1.6 mypasswd mygroup
root:0:0:root
.
.
mylogin:60:40:tech
.
.
```

4. Type **rm mypasswd mygroup** and press **Return** to remove the files created in this activity.
5. Turn to Module 95 (SV), Module 21 (BSD) to continue the learning sequence.

Module 70
kill

DESCRIPTION

The external and internal **kill** command is used to send signals to processes currently executing on the system. Its name is derived from the fact that most signals will cause a process to die, thus it kills the process. Normally **kill** is used in conjunction with the **ps** command, system's, or the PID or ? variable.

COMMAND FORMAT Following is the general format of the **kill** command.

```
kill [ -signal ] PID ...
kill -signal -PGID ...
kill -l
```

BSD (Berkeley)

```
kill [ -signal ] PID ...
kill -l
```

C shell & Korn shell

The C shell and Korn shell have additional formats for the internal **kill** command if job control is supported on the system.

```
kill -l
kill [ -signal ] [ PID... ]
kill [ -signal ] [ %n... ]
```

Options The following options may be used to control how **kill** functions.

-signal	Specifies the *signal* sent to the given PIDs. You can use numbers or the corresponding names. Refer to the following table.

C shell & Korn shell

-l	Displays (lists) the signal numbers and names.
%n	Specifies a job number to kill.

If no *signal* is specified, signal 15 is used. Signal 15 is the signal for software termination, which causes the program to perform internal cleanup, if it is written to do so, and then exit.

Arguments The following arguments may be passed to the **kill** command.

PID ... One or more Process IDs you wish to send a signal.

PGID ... One or more Group Process IDs you wish to send a signal. All processes with the *PGID* group process ID are sent the signal.

0 If the specified PID is zero, then the signal is sent to your login shell and all processes started by it. For example,

```
kill 0
```

sends a software terminate signal to all of your processes, even your shell. The shell catches and ignores the signal. But most other commands are terminated.

FURTHER DISCUSSION The **kill** command is useful for killing runaway or hung background processes. A runaway process is often caused by an infinite loop in a program. In any case the **kill** command can be used to remove a process from the system. A hung process may be caused by a disconnected device, such as a terminal, or printer that has hung. If a background process is waiting for input that will never arrive, you will need to kill the process.

NOTE

Because of the way UNIX handles process signaling, it is possible to have a process hang and not be able to kill it. This is caused by the program not being able to process its signal queue. Thus when you send a signal to the process to terminate it, the process never actually receives the signal. In these situations the system administrator may have to reboot the system to remove the process.

The following table lists the signals you can send to a process. Use the -l option to list out available signals and their corresponding names on your system. The symbolic names and numbers are not the same on all systems. The symbolic names in parenthesis are aliases.

Signal Number	*Symbolic Name*	*Description*
1	HUP	Hang-up. The phone line disconnected.
2	INT	Interrupt. Pressed the Delete key.
3	QUIT	Quit. Pressed the Ctrl-\ key.
4	ILL	Illegal instruction
5	TRAP	Trace trap
6	IOT(ABRT)	IOT instruction
7	EMT	EMT instruction
8	FPE	Floating point exception
9	KILL	Kill process. Cannot be trapped
10	BUS	Bus error
11	SEGV	Segmentation violation
12	SYS	Bad argument to system call
13	PIPE	Write on a pipe with no one to read it
14	ALRM	Alarm clock
15	TERM	Software termination signal (default signal sent from kill)
16	USR1	User-defined signal 1
17	USR2	User-defined signal 2

Signal Number	Symbolic Name	Description
18	CLD(CHLD)	Death of a child
19	PWR	Power fail restart
20	WINCH	Window or mouse signal
21	URG	Urgent socket condition
22	POLL(IO)	Selectable event pending (asynchronous i/o)
23	STOP	Stop process
24	TSTP	Stop process requested from tty (keyboard)
25	CONT	Continue after a stop
26	TTIN	Background process requested tty (keyboard) input; stops process
27	TTOU	Background process requested tty (terminal) output; stops process
28	VTALRM	Virtual timer alarm
29	PROF	Profiling timer alarm
30	XCPU	Exceeded cpu limit
31	XFSZ	Exceeded file size limit

NOTE

For a list of signals handled by your system use the kill -l command. If your system does not support the -l option, then try using the command "grep SIG /usr/include/*/signal.h." Remember to strip off the SIG prefix.

RELATED COMMANDS Refer to the **ps** command described in Module 109.

APPLICATIONS

The **kill** command is used to terminate runaway or hung processes that have been placed in the background. It can also be used from another terminal to terminate an interactive process that has become inoperable for some reason.

When killing application programs it is usually a good idea to terminate the process with a -15 signal first. If the process does not die within a reasonable amount of time (10 to 60 seconds), then you may have to repeat the kill command or send a -9 signal.

TYPICAL OPERATION

In this activity you use the **kill** command to terminate a background command. Begin at the shell prompt.

1. Type the following lines; press **Return** after each line.

```
cj> while true
do
   echo "An infinite loop!"
   sleep 5
done &
3781
```

C Shell

```
cj> while ( 1 )
    echo "An infinite loop!"
    sleep 5
end &
3781
```

This creates an infinite loop that prints a message to your screen, then sleeps for five seconds and repeats itself. Notice that a number was returned to your screen after you pressed Return on the last line. This is the Process ID (PID) of the background process.

2. To terminate the process type **kill** *3781* and press **Return**. Replace 3781 with the number returned after the line ending with &.

   If you started a subshell **sh**, type **exit** and press **Return** to return to your **csh**.

3. Turn to Module 94 to continue the learning sequence.

Module 71
ksh/csh/sh

DESCRIPTION

The **ksh** (pronounced "kay shell") program is a sophisticated command processor. It is also an interpretive programming language. It reads input from the terminal keyboard, interprets the command, and makes requests to the UNIX operating system to perform the requested action. As a programming language the shell lets you store shell commands in a file and execute the file as a program.

The shell used and discussed throughout *Illustrated UNIX* is the enhanced shell written by David Korn. Thus it is commonly called the "Korn shell." The Korn shell is usually stored on the system as **ksh**. For clarity the ksh notation is used.

The **ksh** was added in Release 4 of SV. It is not distributed with standard BSD, although many vendors offer it with their BSD type systems. If you do not have a copy of the Korn shell, you will not be able to use some of the commands discussed in this book. To obtain a copy of the Korn Shell you can contact the AT&T Toolchest Bulletin Board on (201) 974-8317 using a 300 or 1200 Baud modem.

The Korn shell is an enhanced version of the original Bourne shell. It adds such features as:

- aliasing
- functions (later versions of the **sh** support functions)
- command re-entry of commands stored in history file
- in-line editing of history file
- job control (only on systems that support job control)
- tilde substitution
- enhanced variable substitution
- arithmetic expression evaluation
- automatic field padding
- filename completion

C Shell

The C shell is also an enhanced version of the Bourne shell. It was originally written in the late 70s and early 80s. The emphasis was to provide a better interactive shell. The command language was designed to resemble the syntax of the C programming language. The C shell added such features as:

- aliasing
- functions
- command re-entry of commands stored in history file
- in-line editing of history file

- job control (only on systems that support job control)
- tilde substitution
- enhanced variable substitution
- arithmetic expression evaluation
- automatic field padding

The C shell became a popular interactive shell, providing a superior interface alternative to the Bourne shell. It also provided the ground work for the Korn shell.

Discussing the shell is sometimes very confusing. In general UNIX terminology "shell program," "a shell," "shell procedure," "shell script," or "script" refer to a shell program. In this book the term "shell script" will be used to refer to a program written in the shell language.

Because the shell is a large and complex program it has been divided into subsections. Some of these subsections have been made into *modules* and are not included in this module. Most of the *built-in* commands have been placed in a module of their own for easy reference. Built-in commands are *internal* shell commands as compared to external UNIX commands that execute as a separate process. An internal command does not spawn a new process when it is executed.

As you read this module you may find it beneficial to read the module for a specific section, then return to this module to continue the learning sequence.

DEFINITIONS

A *blank* is a tab or space. A *word* is a contiguous set of characters, no blanks allowed. If a tab or space is quoted, then it is no longer a blank; it is a *word*. A *name* is a sequence of digits, letters, or underscores that begins with a letter or underscore. Refer to the Special (Meta)Characters section that follows, and to Module 23, the section "Shell Command Syntax."

COMMAND FORMAT Following is the general format of the **ksh** command.

```
ksh [ ±acefhikmnprstuvx ] [ ±o optargs ] [ args ]
rksh [ ±acefhikmnprstuvx ] [ ±o optargs ] [ args ]
```

C Shell

```
csh [ -cefinstvVxX ] [ arg_list ]
```

Options The following list describes the options and their arguments that may be used to control how **ksh** and **rksh** function.

-a — Automatically export all variables as they are assigned values. Implies that an **export** *Variable* is performed after each variable is assigned a value.

-c *string* — Execute *string* and exit to current shell. String is any command that the shell can execute. Normally, it is the name of a file containing shell scripts.

-e — Exit the shell immediately if running a shell script and an error occurs. If a command terminates abnormally or returns a nonzero return code, then an error occured.

-f	Filename generation is disabled. The shell no longer expands wildcard characters. Refer to Module 54.
-i	Interactive shell. If the standard input and output is connected to a terminal, then the shell is interactive. In interactive mode the shell changes its signal handling. It ignores the TERMINATE and QUIT signals; therefore, a kill 0 or kill 2 does not affect the shell. It also traps and ignores INTERRUPT (kill 1) so you can interrupt the wait command.
-k	Place all keyword arguments in the environment for a command. By default only the keyword arguments preceding the command are exported to the environment.
-m	Background jobs run in a separate process group. Upon completion a line is displayed. The exit status of the job is reported in a completion message. This option is set automatically by interactive shells on systems supporting job control.
-n	Read but do not execute commands. Useful to check for syntax errors.
-o optargs	Allows you to set other and special options. Refer to the set command.
-p	Disables processing of the $HOME/.profile file. The /etc/suid_profile file is used instead of the file defined in the ENV variable. If the real uid(gid) is not equal to the effective uid(gid) this option is automatically set. If you turn this option off, the effective uid and gid are set to the read uid and gid.
-r	The shell runs as a restricted shell.
-s	Commands are read from the standard input; if no arguments remain, then input is switched to the standard input. Standard output is written to the standard error (file descriptor 2).
-t	Read and execute one command, then exit the shell.
-u	Unset variables cause an error when substitution is performed. If a variable has not been assigned a value and it is referenced, then an error will occur.
-v	Verbose. Echo (display) each shell input line as it is read.
-x	Debug. Display command lines as they are executed; each command line is preceded by a +.

C Shell

-c	Same as ksh
-e	Same as ksh
-f	Fast start. Forces the csh not to read the .cshrc file. If the shell is a login shell, .login is also ignored.
-i	Same as ksh
-n	Same as ksh
-s	Same as ksh
-t	Same as ksh
-v	Same as ksh

-V	Sets the *verbose* option before reading the .cshrc file.
-x	Same as ksh
-X	Sets the *echo* option before reading the .cshrc file.

The following examples help clarify some of these options.

```
cj> ksh -x myscript
    # executes the shell script "myscript"
    # displays each line before executing it

cj> START=10 ksh -k myscript INCREMENT=2
    # Assigns keyword parameters before the shell command and after it.
    # The variable "START" is set to 10 and "INCREMENT" is set to 2.
```

The -x can be used to debug shell scripts. The -k option allows you to place keyword assignments after the command has been entered. Without the -k you can only place keyword assignments before the command.

NOTE

The **set** command can be used to change or set options and arguments once a shell has begun execution. Set cannot change or set the -c, -i, -r, and -s options.

Arguments The following list describes the arguments that may be passed to the **ksh** command.

args	Valid arguments are programs or shell scripts that are executable and their arguments. These *args* are referred to as *positional parameters* (see Module 99, Variables/Parameters/Arrays). Parameters may be *keyword parameters*, *constants*, *tilde substitution*, *command substitution*, or *filename generation* output.

INVOCATION If the shell is invoked as **-ksh** or **-rksh**, it assumes to be a **login** shell. A login shell reads commands from the /etc/profile and from either $HOME/.profile or .profile of the current directory if the files exist. If the environment variable ENV is set, then commands are read from the file defined by ENV if it exists. After the shell has finished its login or if it is invoked as a shell script, it reads commands based on the -c, -i, -r, and -s options. Refer to the **Options** section for further details.

C Shell

If a **csh** is a login shell, the commands in the $HOME/.cshrc file are read. The /etc/login file is processed next. The $HOME/.login file is then read. The $HOME/.cshrc file is read each time a **csh** is executed. It is the equivalent of the file defined in the ENV variable of the **ksh**.

Explicit Invocation If **ksh** is the first word of a command, then the shell has been invoked explicitly. This allows you to pass options to the **ksh** program, setting up a new environment to execute a command. The general format of an explicit shell command follows.

```
ksh [ options ] [ command [ options ] [ arguments ] ]
```

The first set of options are the **ksh** options listed in the **Options** section. The second set of options are used by the command and are described in each command-related module.

Implicit Invocation If a shell script is executed, the current shell invokes a new **ksh** program that will read and execute the script. The general format of an implicit shell command follows.

```
command [ options ] [ arguments ]
```

The options on an implicit command are used by the command; they do not control the **ksh** program.

For example, let's say you typed myscript parameter1 and pressed Return. Your current shell would search all directories listed in the PATH variable for a program named "myscript." If it found such a program it would check the type of contents. If the program in the file was considered to be a shell script, your current shell would execute the program by executing the **ksh -c myscript parameter1** command.

RESTRICTED SHELL The restricted shell is used to provide a controlled environment for user names who are not allowed full user access to the system. The restricted **ksh (rksh)** performs the same functions as the normal **ksh**, except the following actions cannot be performed:

- Changing directories (refer to Module 14, **cd**)
- Setting the SHELL and PATH variables to new values
- Using command names or paths containing a /
- Redirecting output (> and >>)

If the restricted shell is used as the login shell, then it reads all the normal setup files. After it reads the $HOME/.profile and the $ENV files it enforces the above restrictions.

If the program requested in a command is a shell script, then **rksh** invokes **ksh** to execute the script. Therefore, the restricted user can be provided with a fully functional shell. Usually, the shell script is an "end-user" type menu that controls what command the user can use. This environment assumes that the user does not have write and execute permissions in the directory that contains the shell script (menu).

This concept of restriction allows the owner of the $HOME/.profile of the restricted user to control the level of restriction. The owner is usually the system administrator, not the user who logged in to the account. The system administrator may set up a restricted bin that contains a limited set of commands that the user can access. The result is a very flexible environment for controlling system users.

SPECIAL (META) CHARACTERS The following list defines the characters that have special meaning to the shell when it scans a command. These characters cause termination of a word unless they are quoted (refer to the **Quoting** section). All of these characters are highly related to commands and must be remembered to understand how commands are interpreted and function.

Special Character	*Description*
;	Sequential execution operator. It functions like a **new-line**; the left-hand command, preceding the ;, must complete before the right-hand command, following the ;, can be executed.
|	Pipe creating operator. A *pipe* is created between the left-hand and right-hand command. The standard output of the left-hand command is "piped" into the standard input of the right-hand command.

&	Background operator. The command preceding the & is executed asynchronously. The shell places the command in "background" mode and continues executing the next command.
|&	Shell Pipe creating operator. The command preceding the |& is executed asynchronously, just as if you had used the &. The difference is the standard input and output of the new command is redirected via a two-way pipe. This two-way pipe is managed by the shell. To write to the pipe, use **print -p** and to read from the pipe, use **read -p**. The command placed in the background must be a complete filter; the first command must read from standard input and the last must write to standard output. See the example below for further explanation.
&&	The logical *and* operator. The right-hand command is executed only if the left-hand command returns a false (nonzero) exit status.
||	The logical *or* operator. The right-hand command is executed only if the left-hand command returns a true (zero) exit status.
(list)	The *subshell* operators. The command enclosed within the parentheses is executed as a subshell or child process of the current shell. Output is combined into one stream.
{list;}	The *grouping* keywords. The command enclosed within the braces is executed in the current shell. Because the braces are keywords, a blank or new-line must follow the first one and a ; or new-line must precede the second one. For example, `{echo This is not valid}` whereas `{ echo This is valid ;}`
space	The space character. The octal 040 or decimal 32 ASCII character. Separates words and names.
tab	The tab character. The octal 012 or decimal 10 ASCII character.
new-line	The new-line character or Return. The octal 015 or decimal 12 ASCII character.

C Shell

|&	Pipe creating operator. A *pipe* is created between the left-hand and right-hand command. The standard output and standard error on the left-hand command is "piped" into the standard input of the right-hand command.

The following examples help clarify how special characters are used.

```
cj> pwd; ls -x              # perform a pwd then list the directory, two commands on
                            # one line
cj> ls | wc                 # count number of entries in directory
cj> sort /etc/passwd > wfile &   # sort the system passwd file and store the output in the
                            # "wfile"
```

cj> cut -f1 -d: /etc/passwd |&	# place in background and connect two-way pipe to # command.
cj> read -p USERNAME	# read first line from pipe
cj> print $USERNAME	# print first line (USERNAME)

COMMAND SYNTAX The command syntax used by the shell is described in Module 23. Module 23 also discusses the syntax of UNIX commands.

COMMENTS A *word* beginning with a # informs the shell that the remainder of the line is a comment. The following forms are valid shell comments.

```
# This is a shell comment

cat phone.db # display the phone database
```

ALIASING Aliasing allows you to assign a command to a name. If an alias has been defined, then the text of the alias (replacement string) is substituted in place of the alias at execution time. If the first word of a command matches the alias, then the substitution occurs. For further explanation of the alias command refer to Module 4.

TILDE (DIRECTORY) SUBSTITUTION The shell checks each *word* for a beginning ~. If a word begins with a ~, the shell reads the *word* up to the next / or blank. It then attempts to match this word with a user name in the /etc/passwd file. If a match is found, then the ~user_name is replaced with the user's HOME directory path. This is referred to as a *tilde* substitution. If ~/ is interpreted by the shell, the $HOME variable is substituted in its place.

Tilde substitution is also performed if a :~ is interpreted or if a ~ is the first character of a keyword value. The general formats of a tilde command follow.

Tilde Substitution	*Description*
~/dir	Reference your $HOME directory.
~user_name/dir	Reference user_name's $HOME directory.
~–	Reference the previous working directory. The value of OLDPWD is substituted.
~+	The absolute path of the present working directory. The value of PWD is substituted.

C Shell

The C shell does not support the ~– or ~+ commands.

The following examples show what they would be converted to by the shell.

cj> cd ~yourlogin	# equivalent to "cd /u1/ts/yourlogin"
cj> cd ~/bin	# equivalent to "cd $HOME/bin" or "cd /u1/ts/mylogin/bin"
cj> cd ~–	# equivalent to "cd /u1/ts/yourlogin"

COMMAND SUBSTITUTION Command substitution is done when the shell reads a command from between two grave accents (‘command‘). The shell executes the command and substitutes the output of the command in place of the ‘command‘. All **new-lines** are removed from

the standard output of the executed command. Command substitution can also be done by using $(command). This syntax makes it easier to nest command substitution commands.

Backslashes (\) used to escape other characters are removed before the command is executed. Backslashes are used to escape grave accents and backslashes. By escaping grave accents you can nest command substitutions.

In the case of a command substitution being enclosed in double quotes ("...`...`..."), a backslash will be left intact unless it is escaping a double quote. For further information on using the backslash to escape characters in command substitution refer to the **Quoting** section of this module.

The following example illustrates how command substitution can be used for different purposes.

```
cj> echo You have `ls | wc -l` files in `pwd`.

cj> DATE=`date "+%D"`

cj> for FILE in `ls`
     do
      echo $FILE.ext
     done
```

NOTE

The 'cat file' command should be replaced by the faster '< file' command. The cat command is invoked as a separate process while the "<" redirection symbol is an internal **ksh** read.

VARIABLES Variables are described in Module 99.

VARIABLE SUBSTITUTION AND REFERENCING Variable substitution or expansion and referencing are described in Module 99.

ARRAYS Arrays are described in Module 99.

BLANK INTERPRETATION The shell scans the command line and separates the *words* into distinct arguments. The words are separated based on internal field separators. The shell's IFS variable defines which characters separate words. The default value of the IFS variable is space, tab, and new-line.

Explicit null arguments, " " or ' ', are retained. Implicit null arguments, arguments that have no value, are removed. The following examples illustrate the implicit and explicit blank interpretation syntax.

```
cj> NUM=

cj> myscript $NUM "" '' "THIS IS THE THIRD POSITIONAL PARAMETER"
```

In the last line the $NUM parameter is removed and the " " becomes positional parameter 1. The " " and ' ' are valid arguments and are passed into the program myscript as positional parameters 1 and 2. The last set of words enclosed in double quotes is passed to myscript as one argument.

FILENAME GENERATION Filename generation is described in Module 54.

QUOTING Quoting provides a way to override special meaning of certain *words* or characters. If a word or character is quoted, then the shell will not interpret its special meaning. There are three types of quoting. The following list describes each type.

Backslash quoting. Each special character may be preceded by a backslash to escape its special meaning. By placing a backslash at the end of a line the new-line is removed; therefore, long lines can be separated into shorter, more readable lines. The problem with using \ is that it becomes cumbersome.

"..." Double quotes. Double quotes allow command and parameter substitution but escape the meaning of all other special words and characters. All text between double quotes is considered one parameter. A backslash quotes the \, ', ", $, and new-line characters within double quotes.

'...' Single quotes (apostrophe). Single quotes provide the most stringent form of quoting. All characters inside single quotes are interpreted with no special meaning. The backslash has no special meaning inside double quotes. Thus it is interpreted as itself, a backslash.

The following examples demonstrate the quoting aspects of the shell.

```
cj> echo "My login shell is $SHELL"
My login shell is /bin/ksh
cj> echo "My login shell is \$SHELL"
My login shell is $SHELL
cj> echo "Spaces          are retained inside double quotes"
Spaces          are retained inside double quotes
cj> echo 'My login shell is $SHELL'
My login shell is $SHELL
```

ARITHMETIC EVALUATION Integer arithmetic is performed by using the internal **let** command. Evaluations are carried out in *long* arithmetic. The **let** command provides faster execution and more understandable syntax than the **expr** command. The **let** command is described in Module 73.

PROMPTING If the shell is used interactively, it prompts you with the value assigned to the PS1 variable. If a new-line is typed and further input is needed to complete the command, the value of PS2 is displayed. The value of the PS3 variable is used as the prompt for the **select** command. If you place an exclamation point in the PS1 or PS2 string, the current event number is displayed. For example, PS1="`hostname`[!]: " produces a prompt like **cj[23]:**.

C Shell

The C shell prompts with the value stored in the *prompt* variable if the shell is interactive. If the shell is not interactive then *prompt* is unset. If an exclamation point (!) is placed in the prompt string, it is replaced by the current event number. For example, set prompt="`hostname`[!]: " produces a prompt like **cj[23]:**.

ENVIRONMENT The shell maintains an environment that consists of named variables and their values. These named variables have been **export**ed or made global by the shell during execution. Any exported named variable is considered part of the environment and can be accessed

by any child process. The **export** and **typeset** commands can be used to export a named variable to the environment. Refer to Modules 47 and 145 for explanations of these two commands.

The -k option causes all named variables to be exported as they are assigned values. Keyword parameters may be used on command lines to export the keyword parameter for the simple command that immediately follows. The -a option allows the keyword parameters to precede or follow the command.

JOB CONTROL Job control allows you to stop, suspend, and start jobs as desired. There are commands to move jobs from background to foreground and vice versa. The shell associates an integer number with each background process. By referring to this number you can manipulate the execution of each job placed in background. For more information on job control, foreground, and background commands refer to the following modules:

Module 9	**bg**	Place job in background to continue execution.
Module 51	**fg**	Place job in foreground.
Module 68	**jobs, %**	List and manipulate suspended, stopped, and executing jobs.
Module 125	**stop**	Stop execution of a background job.
Module 129	**suspend**	Suspend current program execution.

NOTE

Most System V based systems, prior to System V Release 4.0, do not support job control. Nor does the **sh**. The **csh** and **ksh** support job control on systems that have the capability in the kernel. All BSD UNIX systems support job control and a few enhanced System V systems. System V Release 4.0 also supports job control.

COMMAND RE-ENTRY The shell saves each command line you type and send to UNIX for execution in the history file. The **fc** command allows you to list and execute previous commands without retyping the entire command. The history command is an alias provided to list out these commands saved in the history file. The **fc** command is described in Module 50 and the **history** command is described in Module 64.

IN-LINE COMMAND EDITING In-line editing allows you to make corrections to a command you are currently typing or to a previous command stored in the history file. To activate in-line editing you must have the EDITOR variable set to **vi** or **emacs**. You may use the set -o **vi** or set -o **macs** command to select an in-line edit mode.

Once the in-line editing mode is activated you can edit your commands and resend them to UNIX by typing a Return. For some shells to send the command line to UNIX the Return must be a Ctrl-J character instead of Ctrl-M.

The window is a single line that can be scrolled left or right. If text is scrolled off on the left of the screen, a greater than sign (>) appears. If text is scrolled off on the right side, a less than sign (<) appears. The width of the screen can be set by using the COLUMNS variable. The default for COLUMNS is 80.

Emacs Editing Mode The emacs editing mode is based on the **emacs** editor. For further information about editing the history file using **emacs** commands refer to Module 50.

Vi Editing Mode To edit commands saved in your history file simply press the Escape key. You are now in vi command mode. If you want to make a correction to the previous command simply

press k. Now move around on the line using **vi** commands. Make changes to the line then press Return. You don't have to press Esc before pressing Return.

For further information about editing the history file using **vi** commands refer to Module 50.

The **vi** editor is described in Module 151.

INTERNAL COMMANDS There are two types of UNIX commands: *internal* and *external* commands. Internal commands are part of the **ksh**. They are executed within the shell itself. No new process has to be spawned, no code has to be read from disk and loaded into memory, and no PATH searching is performed. Therefore, internal commands are executed much faster than external commands.

Except for the following two internal commands, all internal commands have been placed in a module of their own name or in the Programming Constructs module.

. The *dot* command. The dot command executes a shell script in the current environment. The shell script is read just as if you had typed all the lines at your terminal, without a new child process being started. Usually, a shell script is executed in a subshell (a child process is started) and changes to the environment do not affect the current shell. The dot command circumvents this restriction. Therefore, any environmental changes made in the shell script executed by the dot command will be reflected in the current shell environment.

: The *colon* command. The colon command is the same as the **true** command. It does nothing. It always returns a 0 exit status. It is often used in while loops or if statements to fill a required command position. For example,

```
while :
 do
  echo "Stuck in an infinite loop."
 done
```

is just one of many ways to use the colon command.

C Shell

The C shell uses the **source** command to perform the dot (.) command function used by the **ksh**.

COMMAND EXECUTION The procedures that are followed by the shell to execute a command are described in Module 23.

DIAGNOSTICS AND BUGS Users often have the problem of the shell appearing to be hung or not responding. There are several situations to be aware of and solutions to each.

The first situation is the Ctrl-S and Ctrl-Q dilemma. If you type input on the keyboard and there is no response, first press Ctrl-Q to resume output to your terminal. If this does not work, the system may be extremely slow, in which case you must be patient.

Another possibility is the shell or last program you executed is hung. Try pressing Delete two or three times. If you are in applications, you might try pressing Esc or Ctrl-D depending on the application's required interrupt or attention key.

RELATED COMMANDS Refer to the following modules.

Module	*Module Name*	*Module*	*Module Name*
4	alias	104	popd
9	bg	106	print
14	cd	107	printenv
23	Commands	108	Programming Constructs
38	echo	110	pushd
41	env	111	pwd
42	eval	112	read
44	exec	113	Regular Expressions
45	exit	118	set
47	export	119	setenv
49	false	120	shift
50	fc	121	sleep
51	fg	125	stop
54	Filenames	129	suspend
59	getopts	136	time
64	history	137	times
67	I/O and Redirection	141	trap
68	jobs	142	true
70	kill	145	typeset
73	let	146	ulimit
74	line	147	umask
92	newgrp	152	wait
94	nice	156	whence
96	nohup	157	whereis
99	pack	158	which

RELATED FILES The following list describes the files the **ksh** command uses.

standard input	Reads from the terminal keyboard or the output of a pipe.
standard output	Writes to the terminal screen or the input of a pipe.
standard error	Writes errors to the terminal screen or the input of a pipe.
/etc/profile	Executes the commands stored in this file during login to perform common system login procedures.
/etc/alias	Executes the commands stored in this file during login to set up common aliases.
$HOME/.profile	Executes the commands stored in this file during login to set up user-defined environment.
$ENV	Executed each time a new **ksh** is spawned.
/tmp/ksh* /tmp/sh*	Used for the history file, some implementations use sh* instead of ksh*.
/dev/null	The system trash can, used to dump unwanted output.
/etc/passwd	Used to find home directories for *~name* substitutions.

C Shell

The following list describes the files the **csh** command uses.

standard input	Reads from the terminal keyboard or the output of a pipe.
standard output	Writes to the terminal screen or the input of a pipe.
standard error	Writes errors to the terminal screen or the input of a pipe.
~/.cshrc	Executes the commands stored in this file at start of execution by each new shell.
~/.login	Executes the commands stored in this file during login to set up user-defined environment. Executed after /etc/login.
~/.logout	Executes the commands stored in this file at logout time.
/etc/login	Executes the commands stored in this file after ~/.cshrc.
/tmp/sh*	Temporary file used for the '<<' command.
/dev/null	The system trash can, used to dump unwanted output.
/etc/passwd	Used to find home directories for *~name* substitutions.
/bin/ksh	Used to execute shell scripts that do not begin with a number sign (#) in row 1 column 1.

RETURN CODES The shell returns a nonzero exit code if an error is incurred during noninteractive execution (shell script execution). If the shell is being used interactively, then the exit code of the last command is returned. The value of the exit code is stored in the $? variable.

APPLICATIONS

There are two primary uses of the **ksh** program. The most common is using the shell as a command processor. Where **ksh** reads input from your terminal and processes it as commands. This is referred to as an interactive shell.

The second most common use of the shell is as a programming language. Using the shell as a programming language is beneficial. Shell scripts can be written to store multiple commands in a file. Programming constructs are provided to write a high level language type program. By using the shell, programming projects can be prototyped in a short time frame.

The shell provides a command environment. It is often desirable to change the environment variables for temporary use and then return to the original environment. Because the current shell passes the environment to all child processes, you can start another **ksh** to change your environment. Once the new shell is started you can change the environment, execute the desired commands, and then exit back to your original command processor (**ksh**). Your original environment exists just as you left it.

TYPICAL OPERATION

In this activity you use the ksh (csh) command as a command processor to enter commands from your terminal, an interactive programming language, and to create a new environment to execute commands. Begin at the shell prompt.

1. Type **env** and press **Return**. Notice the ksh's environment is displayed on your terminal screen. Your environment is composed of variables that have been exported and can be passed to child processes.

```
cj> env
LOGNAME=mylogin
MAIL=/usr/mail/mylogin
ENV=/u1/ts/mylogin/.kshrc
HOME=/u1/ts/mylogin
PATH=:/bin:/usr/lbin:/utils:/u1/ts/mylogin/bin:.
TZ=CST6CDT
TMPDIR=/tmp
VISUAL=vi
FCEDIT=vi
SHELL=/bin/ksh
```

C Shell

If you are using the **csh** use **printenv**.

---

2. Type the following text; press **Return** at the end of each line. This is a shell script or program performed in interactive mode. You could store these commands in a file and execute the script as a command.

```
cj> OIFS=$IFS            # save IFS in OIFS to restore later
cj> IFS=":"              # reset the internal field descriptor to colon
cj> cat db/phone | while read LINE
do
  set - $LINE
  echo "$1 \t $4 \t $5"
done
cj> IFS=$OIFS              # restore the original IFS variable
```

The line beginning with **cat** pipes the db/phone into a while loop structure. The **read LINE** reads each line of the file and performs the **set** and **echo** command for each line. The **set** command sets each colon-separated field in db/phone to a positional parameter. The **echo** command then displays selected fields onto your screen. The result is a simple formatted output of the db/phone file.

C Shell

1. There is no easy or reliable way to duplicate this code in the csh. But you can reset the command line arguments using the set command. Simply type the following.

```
cj> set argv=( newarg1 newarg2 newargN )
```

2. Now type **echo $argv[$#argv]** and press **Return**.

---

3. Type **ksh** (csh) and press **Return**. Notice a prompt returns. A new shell has been executed by your login shell. If you do not have **ksh** available use **sh** or **csh**.
4. Type **HOME=/tmp** and press **Return**. Your HOME directory is now /tmp.

C Shell

Type **set home=/tmp**.

---

5. Type **cd** and press **Return**. Your new directory is /tmp.
6. Type **pwd** and press **Return** to display your new HOME directory.
7. Type **export VAR=:** and press **Return**. This exports the VAR variable in the new shell's environment.

C Shell

Type **setenv VAR**.

---

8. Type **env** and press **Return** to display your new environment. Notice the HOME and VAR variables reflect their new values. If you like, continue working in this new environment.

C Shell

Type **printenv**.

---

9. To return to your original shell, type **exit** and press **Return** or press **Ctrl-D**.
10. Type **env** and press **Return** to display your login environment. Notice your HOME variable is set to /u/ts/mylogin and the VAR variable does not exist. Remember, exported variables are passed to child processes but not to parent processes.

C Shell

Type **printenv**.

---

11. Turn to Module 23 to continue the learning sequence.

Module 72
leave (BSD)

DESCRIPTION

The external **leave** command provides a reminder service. You inform **leave** when to notify you to leave. Then **leave** waits until five minutes before that time, then it reminds you to leave. It reminds you again one minute before you should leave. If you are still logged on, **leave** reminds you at the specified time and every minute thereafter. If you have logged off, leave exits before displaying another reminder.

The **leave** command ignores the interrupt, quit, and terminate signals. To remove it from execution, log off the system or perform a **kill -9** *pid*, where *pid* is the process ID for the **leave** process. To get the PID of leave use the ps command.

Some users place the **leave** command in their .login or .profile files without any arguments. This causes **leave** to prompt them for a time during the login process.

COMMAND FORMAT Following is the general format of the **leave** command.

```
leave [ [+]hhmm ]
```

Options The following options may be used to control how **leave** functions.

hhmm Format to specify the time of day, where *hh* represents hours on a 12- or 24-hour clock and *mm* represents minutes. The hours are converted into a 12-hour clock and assumed to be in the next 12 hours. For example, if it is currently 9 A.M. and you specify 0500, **leave** notifies you at 5 minutes to 5:00 P.M.

+hhmm Forces relative time interpretation. Instead of being notified at a certain time, the plus forces **leave** to send you the alarm after *hh* hours and *mm* minutes have passed.

If you do not specify a time option, **leave** prompts you with "When do you want to leave?" If you press Return, **leave** exits immediately. If you reply, **leave** assumes the input is a time and verifies it is valid.

RELATED COMMANDS Refer to the **calendar** and **mail** commands described in modules 11 and 86.

APPLICATIONS

You can use the **leave** command to remind you when you need to leave for important meetings, like lunch with your girlfriend or wife. It is most often used to remind users when to leave to go home. In our frenzy to work 16 hours a day we often forget when to leave our terminals. The leave program is a simple yet helpful reminder service.

TYPICAL OPERATION

In this activity you use the **leave** command to remind you to go to lunch. Begin at the shell prompt.

1. Type **leave 1130** and press **Return**. You may want to change the time you leave for lunch. Now continue with your normal tasks. At 11:25 **leave** will send an alarm to your terminal and again at 11:29. If it is after 11:30 select a later time.
2. Turn to Module 117 to continue the learning sequence.

Module 73
let (ksh) / @ (csh)

DESCRIPTION

The internal **let** command evaluates arithmetic expressions. **let** performs all arithmetic calculations in long integer. No overflow checking is performed on the results.

C Shell

The C shell provides arithmetic evaluation similar to the **let** command. The command line must begin with @ ("at sign").

COMMAND FORMAT Following is the general format of the **let** command.

```
let [ expression ... ]
(( expression ))
```

(( )) The double left parentheses begin the evaluation of an expression. The double right parentheses end the evaluation. This notation is equivalent to the let "expression" command. Double quotes are NOT required to prevent shell interpretation of the *, < and > operators.

C Shell

```
@
@ var = expression
@ var[n] = expression
```

Arguments The following list describes the expressions that may be passed to the **let** command.

expression An arithmetic expression consisting of constants, variables, and operators. The constants and variables must be numbers. Valid formats of expression with **let** format are:

```
let lcv=lcv+1          # no spaces allowed in expression
let "lcv = lcv + 1"    # spaces in expression, must use double quotes
```

Multiple expressions may be placed on the command line.

Variable A shell variable. Assigned the result of the expression or provides a number for the expression. You do not have to precede the variables used in an expression with a dollar sign ($).

Constant	A number or a shell variable containing a number. The variable does not have to be an integer attribute type (see typeset command). The constant is written as *base#number*. The *base* is a number between 2 and 36. If no *base* is given, base 10 is used. Base eight numbers can be represented with a leading 0 (0number) and base sixteen numbers with a leading 0x (0xnumber).
Operator	An arithmetic operator, such as +, –, *, and /. See section on Operators.
( )	Parentheses may be used inside an expression to override the precedence of operators. Basically used to group certain operators with constants to control the order of evaluation.

C Shell

@ *var* = *expression*	With no arguments, the values for all shell variables are displayed.
@*var*[*n*] = *expression*	With arguments, *var* or the *n*th word of *var* is set to the value of expression. If *var*[*n*] is used, it must already exist.

```
@ cnt = $num + 1    # No quotes needed unless the <, >,
                    # &, or | characters are used.
```

The double quotes prevent the shell from interpreting the *, < and > operators. If you use any of these operators, you must enclose the expression in double quotes.

The @ command does not support different base numbers. It uses only base 10.

Operators **let** evaluates the operators in the following order. If two or more operators are in the same expression, then **let** evaluates them left to right on the command line.

()	Override normal precedence rules.
–	Unary minus
!	Logical negation (not)
~	Bitwise negation
* / %	Multiplication, division, remainder. The remainder is that of the first expression after dividing it by the second expression.
+ –	Addition and subtraction
<< >>	Left shift and right shift. Bit shift the first expression by the number of bits specified in the second expression.
<= >= < >	Comparison; less than or equal to, greater than or equal to, less than, greater than
== !=	Equal and not equal

&	Bitwise and. The value contains a 1 in each bit position if both expressions have a 1 in the corresponding bit position. Contains a 0 if either expression has a 0.
^	Bitwise exclusive or. Value contains a 1 in each bit position if only one expression has a 1 in the corresponding bit position. If both expressions have a 1 or a 0, then the value contains a 0 in the corresponding bit position.
|	Bitwise or. The value contains a 1 in each bit position if the corresponding bit of either expression contains a 1. If both expressions contain 0 in corresponding bit positions, then the value contains 0.
&&	logical and. If the first expression evaluates to 0, the second expression is not evaluated.
||	Logical or. If the first expression evaluates to nonzero, the second expression is not evaluated.
=	assignment
op=	Compound assignment. This is the same as *Variable=Variable op expression*, where *op* is *, /, %, +, –, <<, >>, &, ^, or |. For example

```
((CNT += 10))
is equivalent to
((CNT = CNT + 1))
```

C Shell

The C shell supports additional operators and evaluates operators in a different order. The operators are as follows.

```
()                  < > <= >=
~                   == != =~ !~
!                   &
* / %               ^
+ -                 |
* / %               &&
<< >>               ||
```

The operators == and != are the same as **ksh**. The third form, =~, checks for the existence of the left string in the right string. The right string may contain filename expansion characters. The !~ operator is the opposite of =~.

FURTHER DISCUSSION The **let** command can perform assignments and comparisons. Assignments allow you to assign a numeric value to a variable, whereas comparison compares the values. You can use the return code from comparison type commands to control conditional shell programming statements. For example,

```
cj> lcv=1
cj> while (( lcv <= 20 ))
 do
  echo Iteration number ${lcv}
  let "lcv = lcv + 1"
 done
```

displays the number of times a loop has been performed until the lcv variable equals 20.

Double quoted expressions must be separated by a space. For example, the following two lines:

```
cj> let j=j+10 k=k+1
cj> let "j = j + 10" "k = k + 1"
```

are considered identical by **let**.

As described in the Argument section the expression can be composed in various formats. Some of these are discussed in the following examples to clarify their usage. For example,

```
cj> let "num = ( 10 + 90 ) / 5"
```

assigns 20 to num, whereas

```
cj> let "nbr = 10 + 90 / 5"
```

assigns 28 to nbr. To display the values in the two variables type

```
cj> echo $num $nbr
20 28
```

One of the more confusing uses of the **let** command is the *base#number* notation for handling different base number systems. Once you use a base, the variable you used is tagged with the base attribute you gave it with the *base#number* notation. For instance, if you use

```
cj> (( N = 2#1011 ))      # assign binary 1011 ( decimal 11 ) to var N
```

then N is base 2 (typeset -i2 N). If you want to assign base 2 numbers to N, they must be either base 2 variables or you must precede the number with 2#. To add another binary number to N you could type

```
cj> (( N = N + 2#10000000 ))  # add binary 10000000 ( decimal 128 ) to N
```

To display the result, use the **echo** command:

```
cj> echo $N
10001011
```

Once the base has been changed, the variable will always be displayed in that base. You can intermix the bases; for example:

```
cj> (( N = 100 ))
```

assigns decimal 100 to N. When you display N, it is in binary:

```
cj> echo $N
1100100
```

RELATED COMMANDS Refer to the **expr** command described in Module 48 and the **typeset** command described in Module 145.

RETURN CODES The **let** command returns a zero (true) exit status if the value of the expression is nonzero. If the value of the expression is zero, the exit status is 1. If multiple expressions are provided on the command line, then **let** determines its return code based on the last expression.

APPLICATIONS

The **let** command is a replacement for the old method of performing shell arithmetic using the **expr** command. It can be used for general purpose integer math. It performs arithmetic operations 10 to 30 times faster than using the **expr** command. It is commonly used inside looping structures to increment the loop control variable. It may also prove useful when working with arrays. You can perform indexing into your arrays with numbers calculated by **let**, again speeding up your shell script.

TYPICAL OPERATION

In this activity you use the **let** command to control a loop to print the numbers from 1 to 100 on your screen. Using let to increment the count to control a loop is one of the most common uses of the command. Begin at the shell prompt.

1. Type the following lines, pressing **Return** at the end of each line.

```
cj> i=0
cj> while (( ( i = i + 1 ) <= 100 ))
do
 echo $i
done
```

Bourne Shell

```
cj> i=0
cj> while expr $i \<= 100 > /dev/null
do
   i=`expr $i+1`
   echo $i
done
```

C Shell

```
cj> set x=0
cj> while ( $x <= 100 )
echo $x
@ $x=$x + 1
end
```

---

2. Turn to Module 120 to continue the learning sequence.

Module 74
line (SV)

DESCRIPTION

The external **line** command is used to read in a line of input and write it to the standard output. The standard input is usually your keyboard and the standard output is your terminal screen. It always writes a new-line to the standard output. If it reads an end-of-file it returns an exit status of 1.

NOTE

Something to remember about the **line** command: you can only read in a line of text to one variable. The **read** command can assign each word of the input to a different variable. This problem can be resolved using the **set** command.

COMMAND FORMAT Following is the general format of the **line** command.

```
line
```

RELATED COMMANDS Refer to the **echo**, **print**, and **read** commands described in modules 38, 106, and 112.

RELATED FILES The **line** command reads from the standard input and writes to the standard output.

RETURN CODES A return code of 1 is returned if an EOF(Ctrl-D) is encountered.

APPLICATIONS

The **line** command is usually used in shell scripts to read in a line of input and assign it to a variable. To do this you must enclose the **line** command in grave accents to perform command substitution.

NOTE

The **line** command was a substitute command for the shell **read** command before System V Release 2.0. The old **read** command was hardcoded to read from your keyboard. It read from your keyboard even if the input had been redirected from a file. Thus **line** filled a need to read from the standard input even if it had been redirected.

TYPICAL OPERATION

In this activity you use the **line** command to read in a line of input and store it in a variable. Begin at the shell prompt.

1. Type **INLINE=`line`** and press **Return**. Notice your shell prompt did not return.

2. Now type **Input line to the line command!** and press **Return**. The text you typed is stored in the variable named INLINE.
3. To display the contents of INLINE type **echo $INLINE** and press **Return**. The one problem with the **line** command is the assignment of values to variables. Only one variable can be used and it must contain all of the contents read by **line**.
4. Turn to Module 112 to continue the learning sequence.

Module 75
listusers (SV)

DESCRIPTION

The external **listusers** command displays a list of all login (user) names and the user ID contained in the /etc/passwd file. The output is sorted by login name.

COMMAND FORMAT The format of the **listusers** command follows.

```
listusers [ -g groups ] [ -l logins ]
```

Options The following options may be used to control how **jobs** functions.

-g *groups* A comma-separated list of groups. All user logins belonging to the list of groups are displayed.

-l *login* A comma-separated list of logins. All user logins belonging to the list of logins are displayed.

RELATED COMMANDS Refer to the **users** command described in Module 150.

RELATED FILES The **listusers** command reads user information from the /etc/passwd file.

APPLICATIONS

The **listusers** command is useful when you want to know who has an account on the system.

TYPICAL OPERATION

In this activity you use the **listusers** command to list all current users on the system. Begin at the shell prompt.

1. Type **listusers** and press **Return**. You will see a list of users displayed on your terminal.
2. Turn to Module 61 to continue the learning sequence.

Module 76
ln

DESCRIPTION

The external **ln** command links a new filename to an existing file. It is used to provide multiple filenames to the same physical data. It has two formats that allow you to:

- Create an additional filename for an existing file.
- Create corresponding filenames in a different directory for a set of existing files.

NOTE

The link name (new filename) must reside in the same file system. Directory names cannot be linked.

There is another type of link called a "symbolic link." A symbolic link allows you to link filenames across different file systems. It also allows the linking of directory names.

Symbolic links are often referred to as *soft links*. This is because the link is not made using the actual inode number. Instead, it is performed by creating a simple name reference to the existing file or directory.

COMMAND FORMAT The general formats of the **ln** command follow.

```
ln [ -fns ] filename linkname
ln [ -fns ] filename_list directory
```

BSD (Berkeley)

```
ln [ -s ] filename [ linkname ]
ln [ -s ] filename_list  directory
```

Options The options that may be used to control how **ln** functions follow.

-f Force the *ln* to occur. The response of "yes" is assumed and the link is performed. No interactive response is requested. Only works on hard links.

If the standard input is not the terminal keyboard, then the -f option is assumed.

-n If the link name is an existing file, do not overwrite the contents of the file. The -f option overrides this option.

-s Causes a symbolic link to be created. A symbolic link contains the name of the file to which it is linked, unlike a normal link which contains the inode number. Symbolic links may span file systems and refer to directories.

BSD (Berkeley)

-s	Same as SV

Arguments The following list describes the arguments that may be passed to the **ln** command.

filename	The existing file that will be linked to with linkname.
filename_list	The list of files that will be linked to in a different directory.
linkname	The new filename linked to the existing file.
directory	The directory where links are created to each file in the filename_list.

FURTHER DISCUSSION The first format creates an additional filename for an existing file. The linkname is entered into the directory with the same inode number as the existing file.

The new linkname and the existing filename cannot be the same full pathname. They may have the same filename, providing they reside in different directories. If the new linkname is an existing filename, its contents are lost.

BSD (Berkeley)

The BSD version of link does not allow the destruction of existing data. If a filename already exists and you try to use it for a linkname, **ln** complains that the file already exists.

The second format creates a link for one or more files in a specified directory. A link is created in the directory for each file listed in the filename_list.

A link is an entry in a directory pointing to an inode that exists and has a filename. Therefore, no new disk space is used other than the directory entry. Once a link has been created, the two filenames point to the same inode and are equivalent except in their symbolic name.

Before **ln** performs the link, it checks the linkname file for write permissions. If you do not have write permissions on the destination, **ln** will display the mode (**chmod**; Module 17) and request a response from standard input. If the input line you type starts with a "y," **ln** attempts to link the linkname to the existing file. If the ownership permissions allow you to write to the target, the move will occur. If you do not have permission to write to the target, the **mv** will fail.

Some common formats are:

ln personal_ltr pl	# links the "pl" name to the personal_ltr file.
ln memo* ../grp/memos	# links all memo files to directory ../grp/memos

DIAGNOSTICS AND BUGS The **ln** command by default will not link across file systems. This is because file systems may be added or removed at random and inode numbers are only unique within a single file system. Symbolic links circumvent this problem.

RELATED COMMANDS Refer to the **cp**, **mv**, **rm**, **chmod**, **chown**, and **ls** commands described in modules 24, 91, 114, 17, 18, and 84, respectively.

APPLICATIONS

The **ln** command provides an alias for filenames. You can link a short filename to an existing file that has a long filename. **ln** can be used to access files in different directories so full pathnames are not required. It is commonly used by users in the same group to access the same file without having to know the full path.

It can also be used to provide different names to the same program. This way a program can examine the command line to find out what command you wanted and perform the required tasks. For example, **vi** and **ex** are the same executable program, but when the program begins execution it checks the command line. If it finds **vi** was called, then it enters visual mode; otherwise, it enters the line editor mode.

TYPICAL OPERATION

In this activity you use the **ln** command to create a new name for an existing file. Begin at the shell prompt.

1. Type **ln file1 letters** and press **Return**. This will create a new filename in your letters directory named *file1*.
2. To see that *letters/file1* and *file1* are linked, type **ls -i letters/file1 file1** and press **Return**. Notice the numbers to the left of each filename. These are the inode numbers, which should be the same.

```
cj> ls -i letters/file1 file1
1652    letters/file1
1652    file1
```

3. Turn to Module 24 to continue the learning sequence.

Module 77
login

DESCRIPTION

The **login** command allows you to identify yourself to the UNIX operating system. The operating system executes **login** for each new user establishing a terminal session, i.e., trying to log in to the system. You may also execute **login** at your shell prompt to change to a new user name and/or environment.

The **login** command requires two valid inputs; the first is your user name, the second is your password. When logging into the system for the first time you will be prompted for both your user name and your password. If you use **login** to change to a new user name, you can provide the new user name as the first argument followed by environment variables. Login will then prompt for the new password.

COMMAND FORMAT Following is the general format for the **login** command.

```
login [ -d device ] [ user_name [ env_var ] ]
```

BSD (Berkeley)

```
login [ -p ] [ user_name ]
```

Options The following options may be used to control how **jobs** functions.

-d *device* The *device* is taken as the path of the tty port login will connect to if possible.

BSD (Berkeley)

-p Causes all additional environment varaibles to be preserved. By default the preexisting environment variables are discarded.

Arguments The following list describes the arguments that may be passed to the **login** command.

user_name A valid user's login (user) name.

env_vars Environment variables that will be passed to the shell upon a successful login. Variables are described in Module 99.

FURTHER DISCUSSION If you do not type at least one lowercase character at the login prompt, **login** will assume your terminal only supports uppercase and will set the tty port settings to uppercase only.

Depending on the UNIX System and system administrator that you have, the **login** program may have additional restrictions. Some UNIX installations require a dial-up password for modem (telephone line) users. You must type a dial-up password and your personal password correctly to gain access to the system.

A time-out feature may be implemented which will disconnect you from the system if you do not log in successfully within a specified time, usually one minute. The disconnect may be based on a certain number of unsuccessful attempts to log in. If you fail to log in within 5 attempts, the system may disconnect you from the system. If the file /var/adm/loginlog exists, all failed login attempts are logged (SV only).

The environment is initialized to:

HOME=your_home_dir_from_passwd_file
LOGNAME=your_login_name_from_passwd_file
PATH=/usr/bin
SHELL=your_login_shell_from_passwd_file
MAIL=/var/mail/your_login_name_from_passwd_file
TZ=timezone_specification

You can modify the environment by specifying arguments on the login command line or by adding them after your login name when login prompts you for it. The arguments can be in the form AAA or AAA=VVV. Arguments without an equal sign are placed in the environment as

`Ln=VVV`

where *n* is an incremented number starting at 0 for each new variable added to the environment. If the argument contains an equal sign(=) it is placed in the environment unchanged.

If the variable already exists in the environment, the new value overwrites it. The PATH and SHELL environment variables cannot be changed. This prevents restricted shell users from spawning secondary login shells without restrictions.

The backslash(\) character can be used to quote the meaning of characters to login. For example, my\ name would allow a space in the login name.

DIAGNOSTICS AND BUGS The login command may return one of the following messages if it encounters a problem.

login incorrect — If the user name or the password cannot be matched.

no shell — No shell (command interpreter) was found that **login** could execute based on the entry in the /etc/passwd file (field 7). Contact your system administrator.

cannot open password file — The /etc/passwd file is not readable or does not exist. Contact your system administrator.

no directory — No directory was found for your login based on the entry in the /etc/passwd file. Contact your system administrator. Some systems may allow you to log in with / being your HOME directory. It is highly advisable to log out and contact your system administrator.

no utmp entry — The system requires that you **exec login** from your initial login shell instead of just using **login**.

RELATED COMMANDS Refer to the modules 127, 72, 41, and 100 describing **su**, **ksh**, **env** and **passwd**, respectively.

RELATED FILES The following is a list of files **login** uses.

/var/mail/user-name	Mailbox file used to store mail for your user name.
/var/adm/utmp	Used to store accounting information.
/var/adm/wtmp	Used to store accounting information.
/var/adm/loginlog	Used to log failed logins.
/dev/ttyXX	Writes output to specified tty device.
/etc/motd	The message of the day file, displayed during login.
/etc/passwd	The system password file.
/etc/profile	System profile, read by login sh/ksh to set up generic environment.
/etc/login	System login configuration file, read by csh at login.
.profile	Your profile, read by sh/ksh to set up environment.
.login	Your login cofiguration file, read by csh at login.
/var/*/*/*	Path used on System V and some BSD systems.

BSD (Berkeley)

/usr/spool/mail/user-name	Mailbox file used to store mail for your user name.
/etc/utmp	Used to store accounting information.
/etc/wtmp	Used to store accounting information.

APPLICATION

The **login** command is used by the UNIX operating system to control access to the system. It can be used by users already logged on to log in again with a new user name and new environment.

TYPICAL OPERATION

In this activity you use the **login** command to log in to the UNIX System. Locate a terminal that is connected to the UNIX System you wish to use.

1. Type **mylogin** after the login prompt and press **Return**.

```
Your_System_Name login: mylogin
```

NOTE

The # (Del-BSD) key must be used for backspacing and the @ (^U-BSD) key must be used to kill a line of input. You may have to press the "break," "interrupt," or "attention" key so UNIX will recognize your terminal is trying to communicate and at what baud rate. Some systems require you to press Return before the login prompt is displayed.

2. Notice the system responded with a passwd: prompt. Type **imauser2** (your passwd) and press **Return** after the prompt.

NOTE

The password on UNIX System V must be at least six characters long and contain one alpha and one numeric.

3. If you entered the correct user name and password, UNIX logs you into the system. If you entered an incorrect user name or password, re-enter your user name and password as illustrated below.

```
Your_System_Name login: mylogin
passwd:
login incorrect
login: mylogin
passwd:
```

   If you are not successful after several attempts, contact your system administrator for assistance.

4. Turn to Module 127 to continue the learning sequence.

Module 78
logname (SV)

DESCRIPTION

The external **logname** command displays your user name (login name). **logname** returns the value of the environment variable $LOGNAME. The LOGNAME variable is set when you log in to the system.

The **logname** command can be used to perform testing in shell scripts. The user's login name can be checked for access permissions to a shell script. The **logname** output can be used for record keeping within a shell. It is not advisable to depend upon the output of **logname** for security reasons. The user can change the value of LOGNAME, thus changing the output of **logname**.

COMMAND FORMAT Following is the general format of the **logname** command.

```
logname
```

RELATED COMMANDS Refer to *env* and *login* described in modules 41 and 77.

RELATED FILES The LOGNAME variable is set in /etc/profile file on some systems or by the **login** command on others. The **logname** command writes to the standard output.

APPLICATIONS

The **logname** command is used to return the user's login name. It may be used to log which users are using a certain shell script or to limit access to a shell script. If first names are used as login sequences, the $LOGNAME variable may be used to personalize shell scripts.

TYPICAL OPERATION

In this activity you use the **logname** command to display your LOGNAME variable. Begin at the shell prompt.

1. Type **logname** and press **Return** to display the value of your LOGNAME variable.

```
cj> logname
mylogin
```

2. Turn to Module 159 to continue the learning sequence.

Module 79
lp (SV)

DESCRIPTION

The external **lp** command spools output to a printer queue. Spooling is the sending of output to a temporary storage area for later processing by the printer subsystem. The output may be a list of files or the standard input passed to the **lp** command. The **lp** command "wraps" the output into a print request and assigns a print job ID. A print job consists of a control file and the output to be printed.

The features of having a spooling subsystem are provided by **lp**. The following list outlines these features:

- Spool multiple print jobs to printers without waiting.
- Spool jobs to multiple printers.
- Specify the number of copies to be printed.
- Specify options to the driver scripts that perform the actual printing.
- Change the title of the banner page on the printout.
- Notification of your request upon completion via mail or write.

COMMAND FORMAT Following is the general format of the *lp* command.

```
lp [ - ] [ -cmsw ] [ -ddest ] [ -f form [ -d any ] ] [ -H req_type ] \
   [ -nn ] [ -ooption ] [ -P page_list ] [ -q priority_list ] \
   [ -S char_set [ -d any ] ] [ -S print_wheel [ -d any ] ] [ -ttitle ] \
   [ -T type [ -r ] ] [ -y mode_list ] file_list
```

Options The following list describes the options and their arguments that may be used to control how **lp** functions.

- Read from the standard input. This allows you to read from the standard input and from files.

-c Copy files to lp request directory instead of linking. Normally **lp** tries to link to the files if possible. If you do not use the -c option and a link is created to your files, you must not remove any of the files until the request has finished printing. Any changes to linked files will be reflected in the current print request output.

-m Send mail back to you after the files have been printed. Usually no mail is sent upon successful completion of your print request.

-s Suppress response messages from **lp**. Such messages as,

"request id is *printer*-957"

are not displayed on your terminal for each request.

-w Writes a message to your terminal when your print request has completed printing. If you are no longer logged in, mail is sent.

-d*dest* Specify desired destination printer. If your system has multiple printers available for printing, you can select the printer of your choice. The destination *dest* may specify a class of printers. If this is the case, your request will print on the first available printer in the class. If a printer is disabled or no disk space is available to store the request, **lp** will not accept the request for the *dest* printer. If you have the LPDEST variable set, **lp** will use its value for the destination printer. If you do not have LPDEST set and do not specify a destination printer, **lp** uses the system default printer. Set LPDEST by typing LPDEST=*printer* and export LPDEST. If you want this done each time you log in, place these commands in your *.profile* file.

-f*form* [-d any] Print the request on the predefined form named *form*. If the form *form* is not supported on the destination printer, the request is rejected. Forms must be defined by the system administrator using **lpforms**. If **-d any** is used, the request is sent to any printer that supports the specified *form*.

-H*req_type* Print the request according to the *req_type*. Valid *req_types* are;

- hold — Hold the request until resumed. Requests already printing are stopped. Other requests will continue to print.
- resume — Resume a held request. If the request was printing when it was held, it will print next, unless bumped by an immediate request.
- immediate — Print the request immediately, which means after the current request finishes printing. Only available to the lp administrator account.

-n*n* Print *n* copies of the request. If -n*n* is not given, one copy is printed.

-o*option* Specify printer or class dependent options. The *option* is passed to the driver script for the given destination. To specify multiple printer options use the -o*option* notation repeatedly. Your system administrator or the printer vendor sets up the driver scripts. Ask your system administrator for a list of supported options. The default SVR4 interface understands the following options;

- nobanner — Suppress the banner page. The lp administrator can disallow this option at any time.
- nofilebreak — If more than one file is to be printed, do not insert a formfeed between each file.
- length=*n* — Print the request *n* lines long. Where *n* is one of the following;
  - *n*i — Page length of *n* inches.
  - *n*c — Page length of *n* centimeters.
  - *n* — Page length of *n* lines.

  Not usable with the -f option.
- width=*n* — Print the request *n* columns wide. Where *n* is one of the following;
  - *n*i — Page width of *n* inches.
  - *n*c — Page width of *n* centimeters.
  - *n* — Page width of *n* columns.

Not usable with the -f option.

lpi=*n* — Print the request with line pitch set to *n* lines per inch.

Not usable with the -f option.

cpi=*n* — Print the request with character pitch set to *n* characters per inch. Other values for *n* are;

pica	10 columns per inch.
elite	12 columns per inch.
compressed	As many columns as the printer can handle.

Not usable with the -f option.

stty=*options* — A list of valid stty command options. If more than one option is specified, separate with spaces and enclose in double quotes.

-P*page_list* — Print the specified pages. A filter capable of printing certain pages must be available if one is not the request is rejected. The *page_list* can be single pages, ranges(pN-pM), or a combination of both.

-q*priority_level* — Assign current request the specified priority. The *priority_level* can be 0 to 39, with 0 being the highest priority. The default priority assigned to a request is set by the lp administrator.

-S*char_set* [-d any]
-S*print_wheel* [-d any]

Print the request using the specified *char_set* or *print_wheel*. If the *char_set* or *print_wheel* is not supported by the requested printer, the request is rejected. If a form is also specified that requires a different char_set or print_wheel, the request is rejected.

The allowable char_sets and print_wheels are set up by you local lp administrator.

If **-d any** is used, the request is sent to any printer that supports the specified *form*.

-t*title* — Prints the *title* string on the banner page of the printout.

-T*type* [-r] — Print the request on a printer that supports the specified content *type*. If no printers support the content type, the request is filtered into an acceptable type. If no filter is available or the -r option is specified, the request is rejected.

-y*mode_list* — Print the request according to the specified modes. The values of *mode_list* are locally defined. If a specified mode is not supported, the request is rejected.

Arguments The following list describes the arguments that may be passed to the **lp** command.

file_list — One or more files to be printed. All files on the command line are printed as one long request. It may be desirable to use the **pr** command to generate numbered pages with the filename at the top to help distinguish one file from the next.

If no files are specified, the standard input is read and saved in a temporary file for later printing.

FURTHER DISCUSSION The **lp** command is the primary user command of an entire UNIX subsystem, called the line printer (lp) subsystem. The user commands of this subsystem are **lp**, **cancel**, and **lpstat**. Each time you send output to the printer **lp** creates a unique request. The **enable** and **disable** commands are sometimes used by the users for printer control. But they are more administrative oriented, and it depends on your system administrator whether or not you have access to them.

Your print jobs are placed in a print queue as printer requests. The request IDs consist of the destination printer name and a sequence number. For example, if you send your files to the "hplj" printer for printing and the last request was number 1287, then your request ID would be hplj-1288.

The requests are processed by **lpsched**, a background process started when the system is booted. These background processes are referred to as *daemons*.

The **lpsched** daemon checks the queue for each printer and if the printer is not busy (in use or jammed), it processes the next request on the printer queue. The file is sent to the printer and **lpsched** updates the request queue to reflect the change. When the file finishes printing, the request is removed from the queue.

Canceling Request You can cancel a print request by using the **cancel** command described in Module 12.

Printer Status You can check on status of the request queue, the status of the scheduler, and the status of each printer using the **lpstat** command described in Module 83.

RELATED COMMANDS Refer to the **cancel** and **lpstat** commands described in modules 12 and 83.

RELATED FILES The standard input is read if no files are given as arguments. Requests generated by **lp** are printed by **lpsched**.

/var/spool/lp/* Main directory for spooler related files and queues.

APPLICATIONS

The **lp** command is used to print files to the system default printer. You can change the destination printer from the default to any that may exist. Because requests are built and queued by **lp** and the printing is handled by **lpsched** you do not have to wait for your files (output) to be printed. You can continue working while the system prints your job whenever the printer is available.

TYPICAL OPERATION

In this activity you use the **lp** command to print a file on your default printer. Begin at the shell prompt.

1. Type **lp file1** and press **Return**. You see a response from **lp** resembling the following:

```
cj> lp file1
request id is hplj-1703 (1 file)
```

   The name of the default printer in this case is hplj.

2. Print a listing of your HOME directory by typing **ls | lp -w**. Notice a request message is returned to your screen. After your request has completed printing you receive a message on your terminal screen informing you of the completion.
3. Format the /etc/passwd and /etc/group file using the **pr** command before spooling them to the default printer. Type **pr /etc/passwd /etc/group | lp**. The two files are formatted into 66-line pages by **pr** and sent to **lp**. You will learn more about **pr** in a later module.

   Find the physical location of the default printer for your system and pick up your printouts.
4. Turn to Module 12 to continue the learning sequence.

Module 80
lpq (BSD)

DESCRIPTION

The external **lpq** command displays the print jobs spooled in the system printer spool queues. To spool a print job you use the **lpr** command. To remove a job from the spool queue you use the **lprm** command. For each print job submitted by **lpr**, **lpq** displays

- the user's name
- the current rank in the spool queue for the given printer
- the names of the files comprising the job
- the print job number (you can use this number to remove jobs with **lprm**)
- the total number of bytes in the job

Without any options or arguments **lpq** displays information about all print jobs currently in the queue for all printers. If you specify the -P*printer* option, then only jobs queued to the given *printer* are displayed. If you provide a *job#*, then only information pertaining to that job number is displayed. If you provide a *username*, then only jobs spooled by that user are displayed.

COMMAND FORMAT Following is the general format of the **lpq** command.

```
lpq [ +[n] ] [ -l ] [ -Pprinter ] [ job#_list ] [ user_list ]
```

Options The following list describes the options and their arguments that may be used to control how **lpq** functions.

+[*n*]	Displays jobs in the spool (print) queue until the queue is empty. If you supply *n*, **lpq** sleeps for *n* seconds between displays of the queue.
-l	Displays information about each print job in the queue. This is multiline output. Normally **lpq** only displays one line per job. The information is just reformatted on multiple lines.
-P*printer*	Displays print jobs related to the specified printer. Overrides the system default printer and the PRINTER environment variable.

Arguments The following list describes the arguments that may be passed to the **lpq** command.

job#_list	Display queued job information for the specified jobs. The job numbers must be space separated.
user_list	Display queued job information for the specified users. The user names must be space separated.

DIAGNOSTICS AND BUGS Because the spool queue is dynamically updated by **lpr** and **lprm** the information displayed by **lpq** may not be accurate. For instance, if another user sends a print job to a printer at the same time you run **lpq**, you may or may not see the new job spooled to the printer.

The displayed output is padded to accommodate the width of the terminal. Thus exact column positions may change.

NOTE

If **lpq** warns that there is no daemon present, contact your system administrator. He will use the **lpc** command to restart the daemon.

RELATED COMMANDS Refer to the **lpr** and **lprm** commands described in modules 81 and 82.

RELATED FILES The **lpq** program uses the following files.

/etc/termcap	Contains terminal capabilities used to manipulate the screen during repeated displaying.
/etc/printcap	Contains printer characteristics.
/usr/spool/*	The spooling directory. May be reset in /etc/printcap.
/usr/spool/*printer*	The directory for a given printer.
/usr/spool/*/cf*	Control files for print jobs.
/usr/spool/*/df*	Data files for print jobs.
/usr/spool/*/lock	Contains the currently active job number.

APPLICATIONS

The **lpq** command provides a mechanism for viewing the status of a currently queued print job while it is in the system spool queue. You can see how many more print jobs have to be processed before your files are printed. The sizes of each print job is displayed so you can estimate the amount of time for each job to print.

TYPICAL OPERATION

In this activity you use the **lpq** command to display the spool queue. Begin at the shell prompt.

1. Type **lpr /etc/passwd** and press **Return**. This spools a job for you to display using **lpq**.
2. Type **lpq** and press **Return**. Notice the output shows the job you spooled in the previous step. You should probably stop by the printer at your next break and clean up all your printouts.
3. Turn to Module 89 to continue the learning sequence.

Module 81
lpr (BSD)

DESCRIPTION

The external **lpr** command spools print jobs to a specific printer's spool queue for later printing by a printer daemon. The printer daemon provides a mechanism for many users to share one device. The daemon knows when the printer is busy and when it is available. The printer queue is a directory with data and control files which make up a print job. The daemon searches for the oldest control file and sends the data file to the printer device based on parameters in the control file.

The system is set up with a default printer destination. If you do not specify a printer destination, your files are sent to the system default. You can override the system default by defining the PRINTER variable in your shell environment. Typically, BSD uses the **csh**, so the command to set the variable is

```
setenv PRINTER lj
```

For **sh** and **ksh** the command is

```
export PRINTER=lj
```

This will cause your files to be printed on the *lj* printer instead of the system default printer if you do not specify a printer.

If you wish to send a file to a printer other than the system default or your default PRINTER, you simply use the -P*printer* option on the **lpr** command line.

To list out all available printers type **lpc status** and press **Return**. The **lpc** command is used by the system administrator to control the printer daemons and queues.

COMMAND FORMAT Following is the general format of the **lpr** command.

```
lpr [ -cdfghlmnprstv ] [ -#num ] [ -1234 font ] [ -C class ]
    [ -J job ] [ -P printer ] [ -T title ] [ -i[num] ] [ -wnum ]
    [ file_list ]
```

Options The following list describes the options and their arguments that may be used to control how **lpr** functions.

The following options are used by **lpr** to notify the spooling daemon that the input files are not standard ASCII text files. The spooling daemon reads this information from the job control file and uses the appropriate filters to print the data files.

-c Assume the data is output from **ciplot**.

-d Assume the data is in DVI (Stanford) format.

-f Use a FORTRAN filter which interprets the first character of each line as a standard FORTRAN carriage control character.

-g Assume the data is output from **plot** routines. The data is in **plot** format.

-l	Uses a filter which displays control characters as ^X and suppresses form feeds (page breaks).
-n	Assume the data is output from **ditroff** (device independent troff).
-p	Use **pr** to format the data (same as the BSD **print** command).
-t	Assume the files contain data from **troff** (phototypesetter command).
-v	Assume the data is raster image format.

The -c, -d, -f, -g, -l, -n, -t, -v require a special filter to work. The system administrator must define each of these filters in the printcap file.

The following options control how **lpr** performs simple formatting and other functions.

-h	Suppresses the printing of the burst (cover) page.
-m	Sends mail to you upon completion of the printing.
-r	Remove the file after the job has been spooled. If the -s option is used, remove the file after it is has been printed.
-s	Use a symbolic link. Usually files are copied to the spool directory. A symbolic link creates a new file with the same name in the spool directory, which is only a link to the original file.
-#*num*	Prints multiple copies of output. Where *num* is the number of copies of each *file* to print.
-1234 *font*	Specifies a font to mount in position X, where X is 1 through 4. Fonts are in /usr/lib/vfont. Fonts are vendor or site dependent and printer dependent so trial and error is required.
-C*class*	Prints *class* as a job classification on the burst page.
-J*job*	Prints *job* as the job name on the burst page.
-P*printer*	Specifies the destination printer. The output is printed on printer *printer*.
-T*title*	Forces **pr** to use *title* as the head title instead of the *file* name.
-i[*num*]	Indent the output *num* columns. If num is not given, the indention is eight spaces.
-w*num*	Uses *num* as the page width for **pr**.

Arguments The following describes the argument that may be passed to the **lpr** command.

file_list	The list of files to print.
	If no files are listed the standard input is printed.

DIAGNOSTICS AND BUGS If you spool a large file it may be truncated. The size of the file depends on the amount of disk space available in the /usr file system.

Most **lpr** commands will object to spooling binary files but may not complain about data files. Data files usually cause problems for most printers. If the spooler daemon is not running, **lpr** will not place jobs in the spool queue unless you are super-user.

RELATED COMMANDS Refer to the **lpq** and **lprm** commands described in modules 80 and 82.

RELATED FILES The following files are used by **lpr**.

/etc/passwd	Used to identify users
/etc/printcap	Printer capabilities database
/usr/lib/lpd*	Line printer daemons
/usr/spool/*	Directories used for spooling, one for each printer
/usr/spool/*/cf*	Job control files
/usr/spool/*/df*	Data files specified in the "cf" files
/usr/spool/*/tf*	Temporary copies of "cf" files

APPLICATIONS

There is only one reason to use **lpr**. That is to send files or the standard output of another command to a specific printer connected to your system.

You may want to reference Module 105 on **pr** to perform simple page formatting and pipe the output to **lpr**. For example,

```
pr -4 file | lpr -Plj
```

would format the contents of *file* into 4 columns and send the output to the *lj* printer.

TYPICAL OPERATION

You must be using a BSD system or a system that supports the BSD **lpr** command to perform this activity.

In this activity you use the **lpr** command to send a file to the system printer. Begin at the shell prompt.

1. Type **lpr file1** and press **Return**. The file named *file1* is spooled to the default printer. You can use the **lpq** command to check the status of your print job.
2. Type **lpq** and press **Return**. Notice the "Rank" of your job. This informs you how many print jobs are ahead of yours in the print queue. The output resembles,

```
cj> lpq
lj is ready and printing
Rank    Owner       Job  Files                                  Total Size
active mylogin      289  file1                                  199 bytes
```

NOTE

If the default system printer is a high speed printer (1000 lines per minute), your print may not appear in the **lpq** output, because it has already been printed.

3. Type **lpr -p file2** and press **Return** to send the *file2* file to the default printer. After being processed, **pr** provides page headers.
4. Turn to Module 82 to continue the learning sequence.

Module 82
lprm (BSD)

DESCRIPTION

The external **lprm** command removes print jobs from a printer's spool queue. The spool queue is a system controlled directory and normal users cannot directly manipulate the files within the queue. Therefore, to remove a print job you must use the **lprm** command.

By default **lprm**, without options or arguments, removes the currently active print job, assuming the user running **lprm** owns the active print job. If you specify a print job number, **lprm** only removes that specific print job.

The names of all print jobs removed from the queue are displayed to your screen. If no print jobs are removed, no message is returned.

When the super-user specifies a *user_name*, all jobs belonging to that user are removed from the print queue.

COMMAND FORMAT Following is the general format of the **lprm** command.

```
lprm [ -Pprinter ] [ - ] [ job# ] [ username ... ]
```

Options The following list describes the options and their arguments that may be used to control how **lprm** functions.

-P*printer* Specify the printer queue for a certain physical printer. If you do not specify this option, the value of the PRINTER variable is used as the print queue. If PRINTER is not set, then the print queue for the default printer is used.

- Remove all jobs owned by you. If you are the super-user, all jobs are removed from the specified queue.

Arguments The following list describes the arguments that may be passed to the **lprm** command.

job# The number of the print job to remove, which is currently spooled in a print queue

username The name of the user whose print jobs are removed. Only the super-user can specify a different *username*.

DIAGNOSTICS AND BUGS If you receive the following message when using **lprm**, contact your system administrator.

```
lprm: printer: cannot restart printer daemon
```

This message indicates that the **lpd** on your system is not working properly or has failed. This usually requires the system administrator to terminate the printer daemon and restart it or, if it has failed, just restart it.

Depending on printer and system speeds, you may perform an **lprm** that removes the second job in queue by accident. This condition occurs when you have spooled two or more jobs in the print queue back to back. You want to remove the active print job. You run **lpq** to identify the active job; you run **lprm** to remove the job. When you check again, the first two jobs appear to be removed. What actually happened was the first job printed and your **lprm** command removed the second job, which became active. To prevent this from occurring, specify the *job#* you want to remove.

RELATED COMMANDS Refer to the **lpq** and **lpr** commands described in modules 80 and 81.

RELATED FILES The **lprm** command accesses the following files to perform the requests.

/etc/printcap	Printer characteristics database.
/usr/spool/*	Location of printer spooling directories.
/usr/spool/l*/lock	Lock file containing the current daemons PID and the active print job number.

APPLICATIONS

The **lprm** command provides you with the ability to remove print jobs you have decided you no longer need printed, thus saving paper and time for yourself and other users.

TYPICAL OPERATION

In this activity you use the **lprm** command to remove a print job. Begin at the shell prompt.

1. First type **lpr /etc/passwd ; lpq** and press **Return** to queue a job to the default printer and display the queue information. The output from the **lpq** command appears as follows.

```
cj> lpq
lj is ready and printing
Rank    Owner       Job  Files                          Total Size
active mylogin      295  /etc/passwd                    5938 bytes
```

2. Now remove the job by typing **lprm 295** and pressing **Return**. The job will be removed and a message will notify you of the fact. If you are using a very fast printer (1000 lines per minute), the job may print before you have time to remove it. If the job is removed the output of **lprm** resembles the following.

```
cj> lprm 295
dfA295cj dequeued
cfA295cj dequeued
```

3. Turn to Module 80 to continue the learning sequence.

Module 83
lpstat (SV)

DESCRIPTION

The external **lpstat** command returns status about the lp printer subsystem. To learn more about sending output to a printer refer to Module 79. To cancel output sent to a printer refer to Module 12. You can use lpstat to display information or status as follows:

Whether a printer accepts requests or is disabled

- Display classes of printers
- Display the system default printer
- Display all output requests
- Display the status of printers
- Display the status of the lp scheduler
- Display printers and devices
- Display all requests by user

COMMAND FORMAT Following is the general format of the **lpstat** command.

```
lpstat [ -drRst ] [ -a[list] ] [ -c[list] ] [ -f[list] [ -l ] \
       [ -o[list] ] [ -p[list] [ -D ] [ -l ] ]
       [ -S[list] [ -l ] ] [ -u[list] ] [ -v[list] ]
```

The \ informs you to continue the line; no new-line is interpreted.

Options The following list describes the options and their arguments that may be used to control how **lpstat** functions.

-d Displays the system default destination printer.

-r Displays the status of the lp scheduler daemon, **lpsched**.

-R Displays the position of the job in the print queue.

-s Displays a status summary: the system default destination printer, printer classes and their members, lpsched status, and a list of printers and their related devices. Same as using the -c -d -r -v options.

-t Displays all (total) status information. Although the information appears somewhat cluttered, it is in a standard format.

- The first line informs you whether or not the spooling scheduler daemon is running.
- The second line states the default printer.
- The next set of lines maps the printer name to its hardware device name.
- The next set informs you whether or not a printer is accepting print requests.

- The next set displays the current state of the printer queue. The lj-419 request is printing on the lj printer, while the dp printer is idle.
- The last set of information is the print queue.

-a[*list*] Displays the acceptance status of printers in the *list*. If a printer is accepting requests, it is enabled and ready to print. If a printer is not accepting jobs, it is disabled and not ready to print. The administrative commands **enable** and **disable** are used to change this state. The *list* may contain printers and classes of printers. If no *list* is given, all printers in all classes are displayed.

-c[*list*] Displays each printer class name and all printers that are members of each class. The *list* consists of class names. If no *list* is given, all printers in all classes are displayed.

-f[*list*] [-D] [-l] Verifies (by displaying) the list of forms exists. If list is not specified, the default is all. The -l option forces long descriptions for each form in the list.

-o[*list*] Displays the status of output printer requests. *List* contains class names, printer names, and request IDs. If no *list* is given, all requests for printers in all classes are displayed.

-p[*list*] [-D] [-l] Displays the status of each printer. *List* contains printer names. If no *list* is given, all printers are displayed.

-S[*list*] [-l] Verifies (by displaying) the list of character sets and print wheels specified in *list* are supported by the lp print service. The default for list is all. If the -l option is specified, a list of printers that can support the character set or print wheel follows each line.

-u[*list*] Displays the status for each user given in *list*. *List* contains login (user) names. If no *list* is given, only your requests are listed. Possible *list* formats are:

user_name	a user on the local system
sys!*user_name*	a user on system sys
sys!all	all users on system sys
all!*user_name*	a user not on the local system
all!all	all users not on the local system
all	all users on the local system

-v[*list*] Displays the name of each printer and the pathname of the device where the printer is physically connected. The *list* contains printer names. If no *list* is given, all printers are displayed.

FURTHER DISCUSSION The *lists* used in the options can be in one of two forms:

1. A list of items separated by a comma (,) with no spaces around the comma.
2. A list of items enclosed in double quotes separated by commas (,) and/or spaces.

For example, to list requests for users bill and yourself (mylogin), you would type:

```
lpstat -u"bill mylogin"
```

or

```
lpstat -u"bill, mylogin"
```

or

```
lpstat -ubill,mylogin
```

Enabling and Disabling printers Some systems are set up to allow users to use two additional lp subsystem commands. These commands are **enable** and **disable**. They can be used to change the state of a printer.

The **disable** command prevents the specified printer from accepting further requests. If also stops the processing and printing of the current request and any queued requests.

The **enable** command starts the printer on the first request in the queue and allows new requests to be spooled to the printer.

The command formats are:

```
enable printer
disable [ -c ] [ -r[reason] ] printers
```

The -c cancels all currently printing requests on all printers listed. The -r provides a reason why the printer was disabled so lpstat will inform users appropriately. The commands are considered administrative commands by some and user commands by others. Check with your system administrator for how they are used at your site.

RELATED COMMANDS Refer to the **cancel** and **lp** commands described in module 12 and 79.

RELATED FILES The files in the /usr/spool/lp/request directory are viewed to present request status. The /usr/spool/lp/* directory contains information files on printers and devices.

APPLICATIONS

You can use the **lpstat** command to see if your request has been processed. It is useful to see how many requests are queued in front of yours. It can tell you what printers are installed and which ones are available at your site. You may decide to cancel a request and cannot remember the request ID. In this situation use **lpstat** to reveal the request ID.

TYPICAL OPERATION

In this activity you use the **lpstat** command to see what printers are installed at your site and what requests are currently queued for printing. Begin at the shell prompt.

1. Type **lp /etc/passwd** and press **Return**. The **lp** command returns the print request ID as follows.

   ```
   cj> lp /etc/passwd
   request id is lj-419 (1 file)
   ```

2. Type **lpstat -t** and press **Return**. Your screen resembles the following display.

   ```
   cj> lpstat -t
   scheduler is running
   system default destination: lj
   device for lj: /dev/tty001
   device for dp: /dev/tty002
   ```

```
lj accepting request since Aug 15 08:32
dp accepting request since Aug 23 09:58
printer lj now printing lj-419. enabled since Oct 15 12:03
printer dp is idle. enabled since Nov 3 10:23
lj-419          mylogin          5839    Nov 6 07:23 on lj
```

Although the information appears somewhat cluttered, it is in a standard format. The first line informs you whether or not the spooling scheduler daemon is running. The second line states the default printer. The next set of lines maps the printer name to its hardware device name. The next set informs you whether or not a printer is accepting print requests. The next set displays the current state of the printer queue. The lj-419 request is printing on the lj printer, while the dp printer is idle. The last set of information is the print queue.

3. Turn to Module 89 to continue the learning sequence.

Module 84
ls

DESCRIPTION

The external ls command lists the contents of a directory. The ls command displays the names of files and directories. Its options allow you to view additional information for each file or directory. The following functions are provided:

- List filenames and directory names.
- Display the vital inode information about a file.
- Recursively list all subdirectories and their contents.
- Display nonvisual characters of filenames and directory names.
- Multiple formats provided for output.

COMMAND FORMAT Following is the general format of the ls command.

```
ls [ -1CFLRabcdfgilmnopqrstux ] [ file_list ] [ directory_list ]
```

BSD (Berkeley)

```
ls [ -1ACFLRacdfgilqrstu ] [ file_list ] [ directory_list ]
```

Options The following list describes the options and their arguments that may be used to control how ls functions.

-1 Display filenames and directories in single-column format. If the output is to a terminal, ls defaults to multicolumn output. If the output is to a file or a pipe, the output is single column by default. You can use the -1 option to override the multicolumn defaults.

-C Lists output in vertically (down the page) sorted columns. The number of columns depends on the length of the longest filename and the screen width.

-F If the file is a directory, a slash (/) is placed after the name. If the file is marked as executable, an asterisk (*) is placed after the name. If the file is a symbolic link, an at sign (@) is placed after the name.

-L If the file or directory is a symbolic link, list the actual file or directory instead of the symbolic link.

-R Recursively lists subdirectories. Each subdirectory that is encountered is listed in the requested format. If you have four levels of subdirectories below your current directory, ls will list all four levels.

-a List all entries including filenames beginning with a period (.). Without this option ls does not list files starting with periods. If you use the -a option with

the asterisk (*) filename expansion character, the files beginning with a period will be displayed. Without the -a option they would not be listed.

-b Displays nonvisible characters in an octal format. The \ddd notation is used, where ddd is the octal number representing the ASCII character code.

-c Sorts output based on the time of the last inode modification for each file. Mode change, file created, link added, or time change cause inode changes. Changes how the -t and -l options function. The inode is a header containing file information.

-d If the arguments are directories, list only their names. Suppresses the listing of a directory's contents. Often used with the -l option to list the permissions of directories.

-f Interpret each argument as a directory and list its contents. No sorting is performed. The directory is displayed in the order each file or directory was created and inserted into the directory file. If no arguments are given, the current directory (.) is used. The -l, -r, -s, and -t options are ignored. The -a option is assumed.

-g Same as the -l option, except the owner field is not displayed (group only).

-i Precede each filename with its related inode number.

-l List contents of directory in long format. Displays the modes (see following section), number of links, owner, group, number of bytes, time of last modification, and filename for each entry. If the entry is a special file, then the major and minor numbers are displayed in place of the byte count.

-m Display output in stream format. The files are comma separated across the page with no new-lines inserted.

-n Same as the -l option, except the owner and group fields contain the actual user ID and group ID.

-o Same as the -l option, except the group field is not displayed (owner only).

-p If the file is a directory, a slash (/) is placed after its name.

-q Displays nonvisible characters as question marks (?).

-r Reverse the sorting order.

-s The file size is displayed for each entry. The size is in file system blocks. File system blocks are usually 512, 1024, 2048, or 4096. Indirect blocks are included in this size.

-t Sort entries by time stamp. The default is the last modification time.

-u Sort entries by last access instead of last modification. Use with the -t and -l options.

-x List output in horizontally (across the screen) sorted columns. The number of columns depends on the length of the longest filename and the screen width.

If no options are given, each filename or directory name is listed on a separate line.

BSD (Berkeley)	
-A	Same as -a (SV) except the . (current directory) and .. (parent directory) entries are not listed.
-C	Same as SV. If no options are specified -C is assumed.
-F	Same as SV with an additional symbol. If the file is a socket, an equal sign (=) follows it.
-g	Includes the group name in the long listing.
-l	Same as SV except the group name is not listed.
	If no options are given, the -C option is assumed.

Arguments The following list describes the arguments that may be passed to the **ls** command.

file_list One or more files **ls** will search for and attempt to display. Files beginning with a period must be explicitly started with a period or the -a option must be used.

directory_list One or more directories **ls** will search for and attempt to display. Directories may also begin with a period.

If no arguments are given, **ls** displays all of the filenames and directory names of the present working directory. The output is sorted alphabetically.

FURTHER DISCUSSION The default listing produced by **ls** is a single-column list of the current directory. Hidden files (files starting with a period) are not listed. By specifying an option you can change the output format, sorting order, or the files that are listed. If you provide specific filenames and directories as arguments, **ls** will list the filenames first. Each directory and its contents are listed. The options you use are applied throughout the listing.

There are three basic types of output formats:

1. One entry per line (short or long information).
2. Multicolumn formats (sorted down or across).
3. Stream format (all entries on the same output line).

Environment Variables for Multicolumn Format The multicolumn options require knowledge about your terminal screen width. To decide how many columns your terminal screen has, **ls** uses the value of the COLUMNS variable. If COLUMN is not set, **ls** uses the *termcap* or *terminfo* database to retrieve your terminal's screen width. The terminal type defined in the TERM variable is used for the database reference. If the necessary information is not retrieved, then 80 columns are assumed.

Aliases The **ls** command with certain options is often aliased to shorthand names. Some systems are shipped with these commands already available. If you have the Korn or C shells, you can create aliases. If you only have the Bourne shell, you can create shell scripts.

The aliases are:

Korn shell	*C shell*
`alias l="ls -m"`	`alias l ls -m`
`alias ll="ls -l"`	`alias ll ls -l`
`alias lsf="ls -F"`	`alias lsf ls -F`
`alias lsr="ls -R"`	`alias lsr ls -R`
`alias lsx="ls -x"`	`alias lsx ls -x`

The shell scripts are:

```
cj> cat l
exec ls -m $*

cj> cat ll
exec ls -l $*

cj> cat lsf
exec ls -F $*

cj> cat lsr
exec ls -R $*

cj> cat lsx
exec ls -x $*
```

Modes The -l option returns the most information by itself about each file. The first ten characters of the listing produced by the -l option are referred to as the file *mode*. The following is an example of the -l output for one file:

```
-rwxr--r--    1  mylogin    ts        1 Jan 30 07:01 myph
```

File permissions

File type

The mode consists of the file type and the file permissions.

File type The first character of the mode indicates the type of file. The following table defines each possible file mode character:

Character	*File Type*
b	The file is a block special file. A block special file describes a device driver that performs I/O in blocks of data.
c	The file is a character special file. A character special file describes a device driver that performs I/O character by character.
d	The file is a directory. The structure of the file is that of a directory.
p	The file is a FIFO (named piped). A named pipe is used for inter process communications, where two or more processes communicate with one another. (FIFO is an abbreviation for First In/First Out an I/O concept, sometimes referred to as a queue)

The file is an ordinary file. You may store whatever type of data you desire in this type of file.

File Permissions The next nine characters of the mode describe the permissions that allow or restrict users from accessing the file. The nine characters can be separated into three fields of three. The following examples show each field's use:

–rwx———	The read, write, and execute permissions for the **owner**.
——rwx——	The read, write, and execute permissions for **group** members.
———rwx	The read, write, and execute permissions for **other** users.

The first line shows the user read, write, and execute positions as being set. The second line shows the groups read, write, and execute positions as being set. And the third shows the others read, write, and execute positions as being set. The following table defines the possible characters found in the permissions field:

Character	*Definition*
r	Allowed to read the file
w	Allowed to write to the file
x	Allowed to execute the file (program)
l	Mandatory file locking will occur when the file is accessed. The set-group-ID must be enabled and group execution must be disabled.
s	Set-user-ID or set-group-ID execution is enabled (see following description of s and S).
S	Set-user-ID or set-group-ID is enabled but the file is not executable (not a defined mode).
t	Sticky bit and execution are enabled.
T	Sticky bit is enabled but the file is not executable (not a defined mode).

Directory Permissions The permissions on a directory are restricted to the read, write, and execute permissions. The execute permission is interpreted as the search permission. That is, users in the group with the execute permission enabled are allowed to search the directory for files.

Special Modes The set-user-ID, set-group-ID, sticky bit, and file locking modes are special modes providing more flexibility in how UNIX handles the execution of a file.

The set-user-ID is reflected in the user execute position. It allows a different user to assume the user-ID of the file. Thus you execute the file (program) as if logged in as the owner of the file. For example, the following file belongs to the user "luwis:"

```
-rwsr-xr-x   1  luwis    ts    1 Jan 20 17:11 testprog
```

If you execute the testprog file, you will execute it as though the user luwis was executing it. This is useful to provide access to secure data via controlled program environments.

The set-group-ID works on the same principal except when you execute it, the group-ID of the file is assumed, not the user-ID. The set-group-ID is displayed as the group execute position as shown in the following example:

```
-rwxr-sr-x    1  luwis       ts          1 Jan 20 17:11 testprog
```

If either position is occupied by a capital S instead of a lowercase s, you can access the file as though you are the owner but you cannot execute the file.

If the sticky bit is enabled, a lowercase t is placed in the others execute position. The sticky bit informs the system to keep the program text of the file loaded in memory. For example, the following file will be executed. When you exit from it the system will not unload it from memory.

```
-rwxr-xr-t    1  luwis       ts          1 Jan 20 17:11 testprog
```

If a capital T is present, then the file is not executable and can never be loaded into memory. Therefore, it is an undefined permission or state of the file. Only the super-user can set the sticky bit.

File locking is enabled if the group execute position is set to lowercase l. The l represents mandatory file and record locking. If file locking is enabled, a file is temporarily unavailable to all other users while it is being accessed. The following example shows the position of the l in the modes:

```
-rwxr-lr-x    1  luwis       ts          1 Jan 20 17:11 testprog
```

RELATED COMMANDS Refer to the **chgrp**, **chmod**, **chown**, **ln**, **mv**, **rm**, and **find** commands described in modules 16, 17, 18, 76, 91, 114, and 55, respectively.

RELATED FILES The following list of files are used by the **ls** command.

/etc/passwd	The user IDs are matched for the ls -l and ls -o options.
/etc/group	The group IDs are matched for the ls -l and ls -g options.
/usr/share/lib/terminfo/?/*	The terminal definition database for screen control (SV)
/etc/termcap	The terminal definition database for screen control (BSD)

APPLICATIONS

The **ls** command is an essential utility. You use it to know what files exist in your directories. Using options, you can list hidden files (files beginning with a period). Certain options display who owns the file and to which group it belongs. All information about a file can be displayed using the **ls** command.

ls is often used in shell scripts to display the directory for selecting a filename. You may wish to process each file in a loop by using **ls** to perform the list part of a **for** loop.

TYPICAL OPERATION

In this activity you use the **ls** command to list the files in your HOME directory and other directories using different options. Begin at the shell prompt.

1. Type **ls** and press **Return**. Notice that a one-column list of files is returned. If you are on BSD the output will be multicolumn.

2. Now type **ls -CF** and press **Return**. Your screen looks like the following display:

```
cj> ls -CF
/bin    calendar  db/       file1     file2     letters/
```

3. Type **ls -l** and press **Return**. A complete listing is displayed.

```
cj> ls -l
total 1
drwsr-xr-- 2 rjf  ts     24 Jan 24 07:12 bin
-rw-r--r-- 1 rjf  ts    107 Jan 24 08:45 calendar
drwxr-xr-- 1 rjf  ts    512 Jan 24 07:17 db
-rw-r--r-- 1 rjf  ts     70 Jan 24 07:25 file1
-rw-r--r-- 1 rjf  ts    161 Jan 24 07:01 file2
drwxr-xr-- 1 rjf  ts     24 Jan 24 07:12 letters
```

4. Turn to Module 14 to continue the learning sequence.

Module 85
machid (SV)

DESCRIPTION

The external machid commands return a true or false value. Machid is not a command itself but refers to a set of commands. These commands are used to identify the hardware you are using. They are usually used in shell scripts to provide portability. If the machid command you use has a return value of zero (0), you are using the hardware described by the command name. The following example is a typical shell script coding:

```
if vax                  # If on a vax computer,
then cd /usr/lib/vax    # change directories to vax one
else cd /usr/lib/other  # else change dir to other one
fi                      # end the if conditional.
```

COMMAND FORMAT Following is the general format of each machid command.

pdp11	True if on a DEC PDP-11/45™ or PDP-11/70™ computer.
u3b	True if on an AT&T 3B20™ computer.
u3b2	True if on an AT&T 3B2™ computer.
u3b5	True if on an AT&T 3B5™ computer.
u3b15	True if on an AT&T 3B15™ computer.
sun	True if on a Sun Microsystems computer.
sun3	True if on a Sun model 3 computer.
sun4	True if on a Sun model 4 (SPARC™) computer.
vax	True if on a DEC VAX-11/750™ or VAX-11/780™ computer.
u370	True if on an IBM System/370™ computer.

NOTE

Not all vendors implement all of the commands. Usually the vendor only implements the more common hardware types and one for the vendor's own hardware.

RELATED COMMANDS Refer to the **test** command described in Module 135.

RELATED FILES The machid commands do no return output, they simply provide return codes.

RETURN CODES The machid commands return a true (zero) value if you are on the type of machine the command describes. If you are NOT on the machine described by the command name, then a false (nonzero) value is returned.

APPLICATIONS

These commands are used in shell scripts to provide portability across different types of hardware. Shell scripts can be coded to handle different hardware device paths, different command formats, and various other system dependencies. The machid commands are useful in a **make** file to control how a program is compiled. In some cases the **uname** command, described in Module 148, can be used to perform the same task. If **uname** is used, the **test** command must be used to check for specific strings being returned.

TYPICAL OPERATION

In this activity you use the machid commands to display what type of system you are using. Begin at the shell prompt.

1. Type in the following code at your shell prompt. As you enter new lines your secondary prompt (PS2) will be displayed. You may wish to use the **whence** command to see if your system supports any of the commands used in this activity.

```
cj> for CMDTYPE in u3b2 u3b5 vax
do
   if $CMDTYPE
   then echo You are using a $CMDTYPE machine!
   fi
done
```

   You may wish to add additional hardware names on the for line.

2. Turn to Module 144 to continue the learning sequence.

Module 86
mail (BSD) / mailx (SV)

DESCRIPTION

The external **mailx** command is an electronic post office. You can send, receive, and store messages using the **mailx** command. It is based on the old **mail** command but provides a much better interface for handling messages.

You can use the tilde (~) commands to edit, review, and modify your message before sending it to the recipient. You can list, display, save, delete, and pipe messages while in command mode. In general terms **mailx** provides you with a memo/message handler. You can send to one or multiple users. You can reply to the original author or to the author and all other recipients of a message. It even allows you to store messages and replies in files based on user names.

Like many of the UNIX commands **mailx** has been changed and enhanced by most vendors. Therefore, there is no guarantee the information that follows pertains exactly to your system's mail program. The **mailx** information that follows is AT&T System V. The **mail** information is from various BSD based systems. Information common to all systems was used, hopefully providing a complete base of information for the BSD **mail** command.

Throughout the module we refer to the **mailx** command. The BSD **mail** command is assumed to be synonymous unless stated differently.

NOTE

The old **mail** command is not documented in this book. It is slow, limited, and cumbersome to use. The BSD mail is usually located in /usr/ucb/mail. The mail program in /bin/mail is the old version of mail.

COMMAND FORMAT Following is the general format of the **mailx** command.

To Read Mail:

```
mailx [ -deFHIinNUV ] [ -f file ] [ -u user ] [ -T file ]
```

To Send Mail:

```
mailx [ -dFinUV ] [ -h num ] [ -r adr ] [ -s subject ] recipients
```

BSD (Berkeley)

To Read Mail:

```
/usr/ucb/mail [ -inv ] [ -f file ] [ -u user ]
```

To Send Mail:

```
/usr/ucb/mail [ -inv ] [ -s subject ] recipients
```

If you specify user_names, **mailx** attempts to send the message to the list of users. If you do not specify any user_names, **mailx** reads messages from your mailbox or specified file and enters command mode.

Options The following list describes the options and their arguments that may be used to control how **mailx** functions.

System V

-d — Turn on debugging mode. Debug statements are displayed.

-e — Checks for any mail in your mailbox. Nothing is displayed. A return code of 0 (true) is returned if mail does exist. A return code of 1 (false) is returned if no mail exists.

-F — Saves the mail message in a file named after the first user who received the message. This option overrides the internal *record* variable.

-f *file* — Reads mail from the file named *file* instead of your mailbox file (/usr/mail/mylogin, BSD is /usr/spool/mail/mylogin). If you do not specify *file*, the file named *mbox* is used.

-H — Displays only the header summary information and exits from **mailx**.

-h *num* — Sets the number of "hops" a message has made to *num*. A *hop* is when a message is sent from one system to the next via a LAN (Local Area Network) or telephone line. This number is used by network mail delivery software to avoid infinite delivery loops. It is not intended for human use. Refer to the *addsopt* variable in the *Internal Variables* section.

-i — Causes **mailx** to ignore INTERRUPT signals. Refer to the *ignore* variable in the *Internal Variables* section.

-I — Displays the newgroup and article-id header lines in addition to the standard header lines that are displayed. Requires the -f option to be specified.

-n — Causes **mailx** not to initialize from the system default file, /usr/lib/mailx/mailx.rc.

-N — Causes **mailx** not to display the header information during initial startup. By default, **mailx** displays the header line for each message contained in the mailbox file being read.

-r *adr* — Passes the address *adr* to network delivery software. The tilde commands are disabled. This option is used by mail delivery software. It is not intended for human use. Refer to the *addsopt* variable in the *Internal Variable* section.

-s *subject* — Sets the subject field of the message to *subject*. If *subject* contains blanks, you must use quotes. If this option is not used, you are prompted for a subject message each time you send a message to a user.

-T *file* — The message-ID and article-id header lines are saved in *file* after the message is read. Automatically sets the -I option.

-u *user* Causes *mailx* to read messages from *user*'s mailbox instead of your own. The specified user's mailbox must be readable by your userid. If it is read protected, **mailx** will complain and fail.

-U Converts a *uucp* type address to internet standard type address. This option overrides the internal variable *conv*. Refer to the *Internal Variables* section.

BSD (Berkeley)

-f [*filename*] Same as **mailx** -f option
-i Same as **mailx** -i option
-n Same as **mailx** -n option
-s *subject* Same as **mailx** -s option
-u *user* Same as **mailx** -u option
-v Sets verbose mode. Detailed delivery information is displayed. Resembles **mailx** -d option.

CAUTION

The -h, -r, and -U options can be used only if **mailx** uses a mail delivery program other than the /bin/mail program.

Arguments The following list describes the arguments that may be passed to the **mailx** command.

recipients A list of recipients to whom the mail message is sent. A recipient may be one of three types:

user_name The name of a user on the local system or on a remote system. If the user is on a remote system, you must specify how the message is to be sent. The first is the uucp format of system!user. For example, sundal!mitek!rjf sends the mail to the system sundal, which forwards the mail to the system named mitek where the user rjf resides. The second is the internet format of user@system. The same address above is written as rjf@mitek.

shell_command A UNIX shell command that reads the standard input. The recipient name must begin with a | (pipe). The command must be in quotes. For example,

```
mailx rjf "|lp"
```

sends the message to the user rjf and to the default printer.

alias_group An *alias_group* allows you to define a group of users. You then specify the group name and the message is sent to each user in the group. For example, in your *.mailrc* file you might specify

```
alias doc "aba barb ccb jag"
```

so when you specify doc as a recipient all of the users in the "doc" alias_group are sent the message.

OVERVIEW

Since **mailx** can be customized to each user's preferences, we discuss the default behavior of **mailx** in this section. You can customize **mailx** by using shell and internal variables.

Sending Mail When you want to send mail to other users, run **mailx** followed by a list of users. A header is added to the message you are sending. The header identifies you (the sender), the date of creation, and a subject line. The message is then sent to each user's mailbox in the /usr/mail (/usr/spool/mail on BSD) directory. The mailbox is a file named after the user's login or user name. For example, the user *tlp* has a mailbox named /usr/mail/tlp.

Reading Mail When you want to read your mail, run **mailx** without a list of users. **mailx** checks your mailbox for mail. If you have messages in your mailbox, **mailx** displays a header line for each message. You can then perform different commands to manipulate the message list and messages. If you read (or touch) a message, it is saved to the file **mbox** when you exit **mailx**. The **mbox** file contains messages you have received and did not delete or save in a different file. Your *mailbox* (/usr/mail/mylogin) contains all new messages that have not been read.

The **mailx** command is compatible with the older **mail** command and thus uses the same mailbox. The older **mail** program does not include a header for each message, nor the vast array of commands to handle messages.

SENDING MAIL

To send mail you simply specify the users you want to send messages as arguments to **mailx**. For example,

```
mailx bill jill phil
```

sends the message you enter to the users, bill, jill, and phil. When you specify user names on the command line, **mailx** knows you want to send a message to recipients. After you press Return on the command line **mailx** prompts you for a subject line. You can press Return to have **mailx** ignore the subject line or you can enter a line of text. For example,

```
mailx barb
Subject: About the database problem!
```

Only the first 40 characters of the subject line are displayed when the user displays the header information using the *h* command.

After you enter the Subject line, **mailx** enters a simple editor or input mode. You enter your message text line-by-line pressing Return at the end of each line. You can use the Backspace key to erase characters on the current line but you cannot return to previous lines. To end the message you press Ctrl-D at the beginning of a line.

You can use the tilde escape commands to perform special functions while in input mode. One of the tilde commands allows you to enter the editor of your choice to edit the message. So even though **mailx** does not provide a true editor, it does allow you to invoke one to edit your message. For example,

```
mailx barb
Subject: About the database problem!
~v
```

causes **mailx** to enter the visual editor for you to edit your message.

The **mailx** header and tilde escape commands are discussed in the next two sections.

HEADERS When you send a message using **mailx** a header is added to the message. The header contains information much like the header of a standard office memorandum. The header has a Date, From, To, and Subject field. The Date field is automatically set when you invoke **mailx**. It has the same format as the default format returned by the **date** command. The From field is your user_name and your name taken from the /etc/passwd file. The To field contains the names of all users you want to send the message. And the Subject field contains a brief line informing the recipients what the message is about.

This line and all other message header lines are displayed on the recipient's screen when **mailx** is first invoked. The header lines can be displayed at any time from the command mode by using the **header** command. There are environment variables and tilde escape commands you can use to control which parts of the header are displayed.

The **mailx** command also provides an additional field for Carbon Copy control. The *Cc* field can be used by setting the *askcc* variable or by using the *~c* or *~h* tilde escape command. This field is a list of users who are to receive the message. The list of users is displayed in the header so all users know who received the message. A similar field is the *Bcc* (blind carbon copy) field. It works the same as the *Cc* field except the list of users receiving a copy are not listed in the header. So other recipients are not informed of who received the message.

Header Fields The header is composed of several different fields. These fields are:

Field	*Description*
Status	The status of the message: D - deleted message N - new message O - old message R - read message U - unread message > - marks the current message
Date	The date the message was mailed
From	The user who sent the message
To	The user(s) to whom the message is sent
Subject	The subject to which the message relates
Cc	The users to whom carbon copies of the message is sent
Bcc	The users to whom blind carbon copies of the message is sent. Other users are not notified about who received a blind carbon copy.

TILDE ESCAPE COMMANDS The tilde escape commands allow you to edit and modify the message you are sending to the specified users. They can only be used while you are in *insert mode*. You can enter insert mode two ways. The first is by entering the **mailx** command from the shell, followed by user names. Another way to enter the input mode is from the command mode of **mailx**. The *mail* or *reply* commands place you in input mode.

When you are in input mode you can request special tilde escape commands by entering a tilde (~) followed by a command at the beginning of an input line. For example, **~v** followed by **Return**, places you in the **vi** editor, thus allowing you to edit your message with **vi**.

You can change the escape character from tilde (~) to some other character by using the *escape* variable in **mailx**.

The following table describes each of the tilde escape commands.

Escape Command	*Description*
~!*cmd*	Causes **mailx** to invoke the *cmd* command. For example, `~!sh` starts a subshell for you. When you want to return to entering your message, type Ctrl-D. Another example is ~!ls which lists your current directory.
~.	Terminates input mode. The same as Ctrl-D.
~:*mailx cmd* ~_*mailx cmd*	Executes a **mailx** command. You can only use this command if you entered the input mode from the command mode (you were reading mail).
~?	Displays a brief summary for the tilde escape commands.
~A	Inserts the value of the *Sign* variable into your message. Usually *Sign* is set to a signature or sign-off type string. For example, if you had `set sign="Sincerely,"` in your *.mailrc* file and you type ~*A* during input mode, *mailx* inserts "Sincerely" in you text.
~a	Same as ~*A* except inserts the value of the *sign* variable.
~b *name list*	Add the names in *name_list* to the **blind carbon copy** (Bcc) list. The recipients of the message are not informed of who received blind copies of the message.
~c *name list*	Add the names in *name_list* to the carbon copy (Cc) list. The recipients of the message are informed who received carbon copies of the message.
~d	Reads the contents of the *dead.letter* file into your message. See the description of the DEAD variable. The *dead.letter* file is created by **mailx** when it is unable to send the message or it is interrupted. The incomplete message is stored in *dead.letter*.
~e	Invokes an editor so you can edit the message you are sending. The editor used depends on the value of the EDITOR variable.
~f *msglist*	Inserts the messages specified in *msglist* into your current message. This is considered forwarding the message on to another user. You can only use this command if you entered the input mode from the command mode (you were reading mail).
~h	Causes *mailx* to prompt you for the Subject, To, Cc, and Bcc fields. If a field already contains a value, you can edit it as if you just entered the value.
~i *variable*	Inserts the value of the specified variable into your message. For example, `~i Sign` is equivalent to the ~A command. It inserts the value of the "Sign" variable.

~m *msglist*	Inserts the messages specified in the *msglist* into your current message. The inserted message text is indented one tab stop. You can only use this command if you entered input mode from command mode (you were reading mail).
~p	Displays the message you are currently typing.
~q	Simulates a QUIT signal; causes **mailx** to be interrupted. The input session is aborted and **mailx** saves the message to the *dead.letter* file. If no text has been entered, **mailx** does not save to *dead.letter*. See the *DEAD* variable description.
~r *filename* ~< *filename*	Reads the specified file into the message.
~< !*cmd*	Reads the standard output of the command *cmd* into the message.
~s *string*	Inserts *string* into the Subject field.
~t *name list*	Adds the specified names in *name list* to the *To* field.
~v	Invokes the editor defined by the *VISUAL* variable. The default is the **vi** editor.
~w *filename*	Writes the message you have entered so far to the file *filename*.
~x	Exits **mailx** like the *~q* command except the message is not saved to the *dead.letter* file.
~|*cmd*	Sends (pipes) the message you have typed to the command *cmd*. If the *shell_cmd* exits with a return code of 0 (successful), the standard output of the *cmd* replaces the message.

BSD (Berkeley)

The BSD **mail** program does not support the following tilde commands.

~A
~a
~i
~r *filename*
~< *filename*
~< !*shell_cmd*
~x

READING MAIL

To read the mail stored in your mailbox you simply run **mailx** without any arguments. For example,

```
$ mailx
```

If you do not have any mail to read, **mailx** returns a message stating that fact. If you have mail, a numbered list of messages is displayed on your screen and **mailx** enters *command mode*. For example,

```
$ mailx
mailx version 2.2 9/9/88.     Type ? for help.
"/usr/mail/mylogin": 2 messages 2 unread

>N  1  bill      Tue Apr 12 08:31  5/134  "cabling problems"
 N  2  nancy     Tue Apr 12 10:03  4/98   "lunch in the break area"
```

Each line in the list contains the status of the message (New, Old), the sender of the message, delivery time, the size in lines/bytes, and the subject. The current message is preceded by a > (greater than sign). Some of the messages will have subject fields, others may not.

After the list of messages is displayed you are prompted with a ? (question mark). You can type any of the command mode commands to manipulate the list of messages. Some examples are:

?2	Read message number 2.
?q	Quit **mailx**. The read messages are saved to your *mbox* file. Messages that you have not read are left in your *mailbox* file (/usr/mail/mylogin, /usr/spool/mail/mylogin on BSD).
??	Enter a question mark for a brief summary of commands (the help screen).

COMMAND MODE You are placed in command mode when you run **mailx** without specifying recipients on the command line. After the messages are displayed **mailx** prompts you for a command by displaying a ?. At the prompt you can type any of the following commands. The general format of these commands are:

```
[ command ] [ message_list ] [ argument(s) ]
```

If you do not specify a command, the *print* command is assumed. For example, to display the current message you just press Return. To display message number 2 you type 2 and press Return.

If you do not provide a message, the *current message* is assumed. If you do specify a message or list of messages, the specified command acts upon each message. For example,

```
?d
```

deletes the current message. The command

```
?d2
```

deletes message number 2.

Some commands require arguments. You specify the arguments after the command and the message lists.

Message List There are several identifiers you can use to specify a message list. Before we look at the commands, let's step through the possible identifiers you can use in a message list. A *msg* or *msglist* may consist of one or more of the following message identifiers.

Identifier	*Description*
.	The current message. Preceded by a >
^	The first message. Deleted messages are ignored.
$	The last message. Deleted messages are ignored.
*	All messages
n	Message number *n*. The number *n* must be an integer.
n-m	Messages *n* through *m*. The numbers *n* and *m* must be integers.
user	All messages sent to you by the specified *user*.

/*str*	All messages that have the string *str* in the Subject field. The match is not case sensitive (*str* matches uppercase or lowercase equivalents).
:d	All deleted messages
:n	All new messages
:o	All old messages
:r	All read messages
:u	All unread messages

Commands The following table lists all of the commands you can use while in the command mode of **mailx**. Some commands only accept one message while others accept multiple messages. To distinguish between these commands we use *msg* to represent a single message and *msglist* to represent one or more messages. Any commands that use filenames for arguments may have a plus (+) placed before the filename to force the use of the folder variable.

Command	*Description*
!*cmd*	Escapes to the shell and executes the UNIX command *cmd*. After the command completes, control returns to **mailx**. For example, `!ls -x` lists the current directory in column format. Refer to the *SHELL* variable in the ENVIRONMENT VARIABLE section. Also refer to the *bang* variable.
# *comment*	A comment. You may find this useful in the *.mailrc* file.
?	Displays a summary of internal mail commands.
=	Displays the current message number.
a alias *alias name(s)*	Creates an alias string *alias* for a list of user *names*. When you use *alias* as a recipient, all of the users listed are substituted in place of the alias. For example, `alias mfg awk jji` substitutes awk and jji in place of mfg when you send mail to mfg. So the mail is actually sent to awk and jji.
alt alternates [*name(s)*]	Establishes a list of alternate names you use to log in to various systems. When responding to messages using the reply command, these alternate user names are removed from the list of users to whom the message is sent. This prevents you from mailing a message to yourself that one of your alternate accounts previously forwarded. If you do not provide a *name* argument, the current list of alternate user names is displayed.
cd [*dir*] chdir [*dir*]	Changes directory to the specified *dir* directory. If you do not specify a directory, $HOME is used.

co [*msglist*] [*file*]
copy [*msglist*] [*file*]

Copies the messages you specify in *msglist* to the given *file*. The messages are not marked as being saved. If you do not specify a *msglist*, the current message is copied. If you do not specify a *file*, the messages are copied to *mbox* unless you have defined the MBOX variable to a different filename.

C [*msglist*]
Copy [*msglist*]

Copies the messages to a file whose name is derived from the sender of each message. The messages are not marked as being saved. An example,

```
? Copy 1
```

saves message 1 to the file *bill* since the message was sent to you by the user bill.

d [*msglist*]
delete [*msglist*]

Deletes the messages specified in the *msglist*. The messages are removed from your system *mailbox*. If you have the *autoprint* variable set, the message following the last deleted message is displayed.

di [*header_field(s)*]
discard [*header_field(s)*]

Suppress the specified fields of each message header. For example,

```
di status
```

causes the status field not to show up when you use the h command. Refer to the section of Header Fields for a list of header fields you may specify.

dp [*msglist*]
dt [*msglist*]

Deletes the specified messages in *msglist* and displays the next message that follows the last message in *msglist*.

ec *string*
echo *string*

Echo the given string. Functions like the UNIX **echo** command. Used in the *.mailrc* file.

e [*msglist*]
edit [*msglist*]

Edit the messages you specified in the *msglist*. Each message is copied to a temporary file and edited. The name of the editor is the value of the *EDITOR* variable. If *EDITOR* is not set, the *ed* editor is used.

ex
exit

Exits from **mailx**. The *mailbox* file is not changed. No messages are saved to the *mbox* file.

fi
file [*file*]

Switches *mailbox* files from the current file to the new mailbox *file*. For example,

```
file mbox
```

causes mailx to quit the current file and read in the *mbox* file. If you do not specify a file, your system default file (/usr/mail/mylogin) is used. The *file* name may consist of one of the following special characters.

%	Your current *mailbox*

(/usr/mail/mylogin).

%user — The *mailbox* belonging to *user*. You must have read access to the specified user's *mailbox* file.

— The previous file you were viewing messages from.

& — Your *mbox* file.

fold
folder *file* — Same as the *file* command.

folders — Displays the name of each file contained in your mail directory. A folder is synonymous to a file. Each folder contains messages to and from a user. You can specify the name of your mail directory using the *folder* variable.

fo
followup [*msg*] — Sends your follow-up message to the author of the *msg*. A copy of the message you send is kept in a file named after the author of the original message. Ignores the value of the *record* variable.

F
Followup [*msglist*] — Sends your follow-up message to the author of each message in the *msglist*. Your response is to the first message in the message list. A copy of the message you send is stored in a file named after the author of the first message. The value of the *record* variable is ignored.

f
from [*msglist*] — Displays the header information for the specified messages.

g
group *alias name(s)* — Same as the *alias* command.

headers [*msglist*] — Displays the header lines for the specified messages. The number of header lines displayed at one time can be controlled by the *screen* variable or the *z* command.

hel
help — Displays a brief summary of the more popular **mailx** commands.

ho
hold [*msglist*] — Stores (holds) the specified messages in your *mailbox* (/usr/mail/mylogin).

i
el
en — See if, else, end that follow.

if *mode*
mail_cmds
else *mail_cmds*
endif — The send or receive if else command. The *mode* is either *s* (send) or *r* (receive). If the mode is set to *s*, the commands up to the else or endif (if the else is not provided) are executed. The commands after the else and before the endif are executed if **mailx** is in the receive mode. If the mode is *r* then the conditions are reversed. For example,

```
if s
  pipe . lp
else
  discard cc bcc
endif
```

might be used in the *.mailrc* file to cause all messages you send to users to be piped to the **lp** command. If the messages are being read, then the *cc* and *bcc* fields are discarded from the header lines.

ignore [*header_field(s)*]
: Same as the discard command.

l
list
: Displays a list of all command mode commands you can use while in **mailx**. No explanation is given for the commands. Helps jar your memory though.

m
mail *name(s)*
: Allows you to mail a message to the specified user(s). **mailx** enters the send mode as if you had specified a list of names as arguments on the **mailx** command line.

mb
mbox [*msglist*]
: Causes the list of mail messages, *msglist*, to be copied to the *mbox* file when you exit **mailx** using the *quit* command. You can specify the name of the *mbox* file by setting the value of the *MBOX* variable to the desired name.

n
next [*msg*]
: Search to the next message matching the specified *msg*. Refer to the message list section preceding these commands. You use this command to search forward for a message sent by a particular user or a message containing a particular string.

|
pi
pipe [*msglist*] ["*cmd*"]
: Pipes the messages you specify in *msglist* to the standard input of the UNIX *cmd* command. For example,

```
pipe . "lp -dlaser"
```

sends the current message (.) to the UNIX **lp** command, with a destination printer of laser. If you do not specify a message, the current message is sent. If you do not provide a *cmd* command, the value of the cmd variable is used. If you have the *page* variable set, a form feed is inserted between each message. The *pipe* command flags the message as being read, which affects how other commands perform.

pre
preserve [*msglist*]
: Same as the hold command.

p
print [*msglist*]

Displays each message to your screen. You can set the *crt* variable to control how many lines you want displayed per "page." The paging command defined by the *PAGER* variable is used to page through all of the messages. The default value for *PAGER* is the **pg** (**more** on BSD) command.

P
Print [*msglist*]

Same as the *print* command except the header fields are displayed even if the *discard/ignore* command has been used to suppress header fields.

q
quit

Quits **mailx**. Messages that you have read are saved to the *mbox* file. Messages that you have not read are returned to your *mailbox* (/usr/mail/mylogin). Messages marked for deletion are removed from your *mailbox* and are not saved to any file.

r
reply [*msg*]

Allows you to enter a reply to the author of the specified message. The subject line of the original message is used in your message. Your reply message is also sent to all the other recipients of the original message. If the *record* variable is set to a filename, then your reply message is saved in the file.

R
Reply [*msglist*]

Allows you to send a reply to the sender of each message specified in *msglist*. The subject line from the first message is used in your message as the subject line. A copy of your response message is saved in the file specified by the record variable. The *Reply* command is like the *Followup* command except *Followup* saves your response in a file named after the author of the original message, whereas *Reply* saves your response to a general response file or does not save the file at all.

respond [*msglist*]

Same as the reply command.

Respond [*msglist*]

Same as the Reply command.

Save [*msglist*]

Saves all of the messages in the *msglist* to a file named after the sender of the first message. Any network addressing is removed from the sender name.

save [*file*]
save [*msglist*] [*file*]

Saves the indicated messages to the *file* you specify. The first form saves the current message to the *file*. If you do not specify a file, the mbox file is used. Messages you save to a *file* are deleted from your mailbox unless you have the *keepsave* variable set.

set

Displays all internal variables and their values you have set. You can set these variables in the *.mailrc* file or in command mode using the set command. See the Environmental Variables section that follows for a full description of each variable.

set *name*

Set the Boolean variable to true. Use the *unset* command to set the variable to false. For example,

set bang
sets the bang variable to true.

set *name=value* — Set the variable to a specified numeric or string *value*. You can create variables of your own, then retrieve the value by using the ~i command.

sh
shell — Invokes a subshell. By default the UNIX **sh** shell is invoked. You can specify the shell of your choice in the *SHELL* variable. To return to **mailx** after entering a subshell you can type exit and press Return or press Ctrl-D.

si
size [*msglist*] — Displays the size of each message in number of characters.

so
source *file* — Reads the *file* and executes each line as a **mailx** command. After the file has been processed, control returns to your current command mode session. An example would be if you made changes to your *.mailrc* file while in **mailx** and you want the changes to take effect without having to exit and re-enter **mailx**.

to
top [*msglist*] — Displays the first five lines of each message you specify. You can change the number of lines to any number you prefer by setting the *toplines* variable.

tou
touch [*msglist*] — "Touches" each message you specify to be saved to *mbox*. Makes **mailx** consider the messages to have been read.

type [*msglist*] — Same as the print command.

Type [*msglist*] — Same as the Print command.

u
undelete [*msglist*] — Changes the status of deleted messages back to normal. You can only change the status of a message you have deleted in the current session of mailx. Once you exit **mailx** you cannot undelete a message you deleted in an earlier session. If you have the *autoprint* variable set, the last message in the *msglist* is displayed when you perform the *undelete*.

uns
unset *names(s)* — Unsets or removes the value of the indicated **mailx** variables.

ve
version — Displays the current version of the **mailx** you are using and the date it was released. Handy for those who just want to know and useful if you need to report problems to your vendor.

vi
visual [*msglist*] — Invokes the editor specified by the *visual* variable. Each file in the *msglist* is edited. The default editor is **vi**.

w
write [*msglist*] *file* — Writes the specified messages in *msglist* to the *file*. The header information and trailing blank line are not copied to the *file*.

x
xit — Same as the *exit* command.

z[+] — Scrolls the list of header lines one screen forward. The *screen* variable may be set to control the number of lines displayed in a screen.

z- — Scrolls the list of header lines one screen backward. The *screen* variable may be set to control the number of lines displayed in a screen.

BSD (Berkeley)

The BSD **mail** program does not support the following commands.

followup	list
Followup	pipe (|)
discard	Save
if s | r	touch
mail_cmds	version
else	
mail_cmds	
endif	

The BSD **mail** does, however, support the following commands that are not supported under System V **mailx**.

retain *header_list* — Opposite of the *ignore* command. It adds the given header names to the list of headers to retain. When **mail** displays the header fields only those in the retain list are displayed. With no arguments the list of retained header fields is displayed.

U
unread [*msglist*] — Mark each message in the given list as unread.

ENVIRONMENTAL VARIABLES

You can set shell variables and internal **mailx** variables to control how **mailx** functions. Shell variables must be set before invoking **mailx** and cannot be changed from within **mailx**. Internal variables may be set while you are in command mode or in the *.mailrc* file.

SHELL VARIABLES The following variables are normal shell variables. You might want to set them in your *.profile* (.login or .cshrc-csh) file so they are always set correctly when you log in. For further information about shell variables refer to Module 99. You cannot change the value of these variables while inside **mailx**.

Variable	*Description*
HOME=*directory*	Defines your home directory. Your home directory is usually the same as your login directory (/u1/ts/mylogin).
MAIL=*filename*	The path where the initial mail is read. The default is /usr/mail/*user* on SV and /usr/spool/mail/*user* on BSD.
MAILRC=*filename*	Defines what file should be read by **mailx** during startup. If you do not define this variable, **mailx** looks for a file named $HOME/.mailrc.

INTERNAL VARIABLES These variables are known only to the **mailx** command. You can *set* and *unset* them in the *.mailrc* file or at the command mode prompt. There are two basic types of variables, Boolean and assignment.

Boolean Variables The first type is called Boolean. These variables have the value of true or false. If they are set they have a true value. If they are not set they have a false value. For example,

```
set askcc
```

sets the askcc variable to true. Thus after you have entered your message to send, **mailx** prompts you for a Cc list.

If a variable is set to true, it is referred to as a normal name. If it is unset to false, its name becomes *no*varname. For example,

```
set allnet
```

sets allnet to true. If you use the unset command to set the variable to false, the name changes to *noallnet*.

```
unset allnet
```

Assignment Variables The second type of variables are called assignment variables. That is, you can assign values to the variable. For example,

```
set cmd="lp -dlaser"
```

sets the internal *cmd* variable to the string "lp -dlaser." So each time you use the *pipe* command, the messages are sent to the "lp -dlaser" command.

Variable Table The following table describes the internal variables recognized by **mailx**. The Boolean variables are listed by name only. Assignment variables are listed with an equal sign (=) and a generic type value required.

Variable	*Description*
addsopt	True by default. It informs **mailx** to use the /bin/mail command to deliver messages. You can use the *noaddsopt* variable if you do not want to use the /bin/mail command. If you do not use /bin/mail, you cannot use the -h, -r, or -U options.

allnet	False by default (noallnet). Causes **mailx** to match all messages with the same login name as identical. This ignores the network address part of the message name. For example, the message names
	`moovax!john`
	`sunup!john`

are treated as equal names in a message list.

append — False by default (noappend). When true, causes **mailx** to append messages to the *mbox* file when you exit from read mode. The default is to insert the messages at the beginning of the *mbox* file.

askcc — False by default (noaskcc). When true, **mailx** prompts you for a list of users you want to send a Carbon copy (Cc) of the message.

asksub — True by default. When true, **mailx** prompts you for a Subject line when you send a message. If you do not type any text and just press Return, no subject is sent.

autoprint — False by default. When true, displays the last affected message automatically after you have used the delete or undelete command.

bang — False by default. When true, exclamation marks (!) used in shell escape commands are interpreted the same as in **vi**.

cmd=*cmd* — No default. Sets the cmd variable to the UNIX command *cmd*. A common *cmd* might be "lp -dlaser."

conv=*conversion* — Sets the method used to convert address formats. The only conversion supported at this time is *internet*, which converts uucp addresses to internet addresses. The uucp address format is system!user. The internet format is user@system. The internet format requires a mail delivery program to conform to the RFC822 standard for electronic mail addressing.

crt=*number* — If set to a number, causes messages containing more than *number* lines to be piped through the command defined by the *PAGER* variable. The *PAGER* variable is set to **pg** by default.

DEAD=*file* — Partial messages from when **mailx** was interrupted and undeliverable messages are stored in the specified *file*. The default file is *dead.letter*.

debug — False by default. When true, causes **mailx** to display verbose diagnostic messages for debugging. No messages are delivered if the debug variable is set.

dot — False by default. When true, causes a line beginning with a single dot (.) to terminate input when you are entering a message to be mailed.

EDITOR=*cmd* — Defines which editor is invoked when you use the tilde escape command ~*e* or the command mode command edit. The default value is **ed**.

escape=*c* — Sets the tilde escape character to character *c*. The default value of *c* is ~ (tilde). For example, if you set escape=%, then you would type %v to invoke the visual editor instead of ~v.

folder=*directory* — Defines which directory is used for saving mail files. To have files saved to this directory you must precede them with a + (plus sign). Suppose you have set folder=mail. Then you issue the following command,

```
s +proj1
```

which saves the current message to the file named proj1 in the folder directory $HOME/mail ($HOME/mail/proj1). The directory name is assumed to be relative from your $HOME directory unless you begin it with a /. For example, folder=mail sets the directory to $HOME/mail. But folder=/tmp, sets the directory to /tmp.

If you want to reference files in your folder directory from the UNIX command line, you must define a shell variable. For example,

```
export folder=mail
```

sets the shell variable *folder* to mail. This must be done to perform commands such as

```
mailx -f +proj1
```

No default value exists for folder.

header — True by default. When true, causes the first page of header lines to be displayed when you start mailx in read mode.

hold — False by default. Saves messages you have touched (read) in your *mailbox* (/usr/mail/mylogin) instead of your *mbox* file.

ignore — False by default. When true, causes **mailx** to ignore interrupt signals while in insert mode. This prevents **mailx** from saving the partial message to the *dead.letter* file and then aborting. It comes in handy when you are dialed in on a noisy line. A noisy phone line generates strange characters because of noise interference. It is not uncommon for these characters to be Breaks and Interrupts.

ignoreeof — False by default. When true, the end-of-file signal (Ctrl-D) is ignored during input mode. This is also handy for noisy dial-up lines. If you do set this variable to true, you must either use the ~. command or set the dot variable and use a single . to terminate input.

keep — False by default. When true, your *mailbox* (/usr/mail/mylogin) is kept even when empty. Normally, your *mailbox* is removed if it becomes empty.

keepsave — False by default. When set, a copy of a message is kept in your *mailbox* (/usr/mail/mylogin) even if you have saved the message to another file.

LISTER=*cmd* — Defines the UNIX command *cmd* to be used when you use the *folders* command to list files in the folder directory. The default *cmd* command is **ls**. You might want to try setting **LISTER="ls -x"**.

MBOX=*file* — Specifies the *file* to save messages that have been touched (read) when you *quit* **mailx**. If you save files while in **mailx**, this variable is ignored. If you exit via the *xit* command, messages are not saved to the *MBOX* file. The default value of MBOX is *$HOME/mbox*.

metoo — False by default. If false, your name is removed from the list of recipients when you use the reply command. If true, your name is left in the list of recipients. Thus you receive a copy of the reply message.

onehop — False by default. When true, the *reply* command does not adjust return addresses based on the address of the first recipient. Instead, the reply message is sent to systems only one hop away. For example, let's say sunrise!tina sends mail to moovax!nancy and sunup!bill. If bill replies (using the reply command), his reply message is sent to sunrise!tina and sunrise!moovax!nancy. If onehop is set, then the replies are sent to sunrise!tina and moovax!nancy. You should only set onehop if you are on a local network where all systems can communicate with each other directly.

outfolder	False by default. When true, all messages saved with the *Save*, *Copy*, *followup*, and *Followup* commands are stored in the *folder* directory. The *record* variable must be set. If you specify an absolute path (a path beginning with a /), this variable is ignored.
page	False by default. When true, causes a form-feed character to be inserted between each message passed to a *pipe* command.
PAGER=*cmd*	Defines the UNIX command *cmd* to be used for paging through messages if the *crt* variable is set. The default *cmd* is **pg**.
prompt="*str*"	Defines the command mode prompt to be *str*. The default prompt is set to a ? (question mark blank).
quiet	False by default. When true, the beginning **mailx** message and version information is suppressed.
record=*file*	False by default. When set, causes all outgoing messages to be saved in *file*.
save	True by default. Causes messages that were interrupted to be saved in the *dead.letter* file. Refer to the DEAD variable.
screen=*number*	Defines the number of lines to be displayed by the header command.
sendmail=*cmd*	Defines the command used to send (deliver) mail. The default is **mail**. Most users don't worry with this variable.
sendwait	False by default. If true, **mailx** does not place a process in background when sending mail. Instead, your message is sent in the foreground. Control of your terminal is returned to you when the mail message has been delivered. May cause **mailx** to be extremely slow, depending on your system and the address resolution required.
SHELL=*cmd*	Defines the UNIX command to run when you use the ! or ~! commands. The default value is **sh**.
showto	False by default. When true, this variable causes the name of the recipient to be displayed in the header line instead of the sender. For example, if you send a message to Bill and a carbon copy to yourself, the message header will display your name because you're the sender. But if you set showto, then Bill's name would be displayed in the header line.
sign=*str*	Contains the string *str* that is inserted into your message when you use the *~a* command. Contains no default value. Common use is to assign a signature or header to the variable.
Sign=*str*	Contains the string *str* that is inserted into your message when you use the *~A* command. This variable contains no default value. Also used for signature or header type strings.
toplines=*number*	Defines the number of lines the *top* command uses. The default value is 5.
VISUAL=*cmd*	Defines the screen editor to use when you use the *visual* or *~v* command. The default value is **vi**.

BSD (Berkeley)

The BSD **mail** program does not support the following variables.

allnet	outfolder
asksub	page
autoprint	prompt
bang	save
cmd	screen
conv	sendmail
DEAD	sendwait
header	showto
keep	sign
keepsave	Sign
onehop	SMARTMAILER

The BSD **mail** does, however, support the following variables that are not supported under System V **mailx**.

ask	Same as System V *asksub* variable
nosave	Same as System V *save* variable
verbose	Same as the *-v* option. The actual delivery of mail is performed in foreground and displayed on your screen, much like the *sendwait* variable in System V.

REMOTE MAIL You can use **mailx** to send mail to users on remote systems. These systems must be connected via a direct wire, a modem, or a LAN. If you are connected to remote systems, you can use the *uucp* address format or the Internet (Arpanet) format, depending on your site's setup. For information on how your system is connected to other systems, contact your system administrator. You can use **mailx** to send the system administrator mail by typing **mailx root**.

If you have a *uucp* connection to other systems, you can use the following address format:

```
mailx host!name
```

where *host* is the name of the system and *name* is the name of the user you wish to send the message. This format is for a system directly linked to your system. To send to a system via "hops" (using other systems to get to the destination) you must specify each system in the path. For example,

```
mailx mitek1!mitek3!mitek5!nancy
```

would send the message to the *mitek1* system, which would forward the message to the *mitek3* system. The *mitek3* system would forward the message to the *mitek5* system, which would deliver the message to the user *nancy*.

If you are connected to the ARPANET or you have the network MAP for uucp, you can use the format:

```
mailx nancy@mitek5
```

to send the same message to the same user on *mitek5*. This is a much shorter and more convenient form of a remote address. But it does require a map of all systems you wish to send a message.

For further information on using remote mail at your site, contact your system administrator.

USING FOLDERS The **mailx** command provides a way to organize your mail. The feature is called *folders*. This feature allows you to save messages in a certain directory. Within this directory you can save messages to files based on the names of recipients, senders, or subjects.

There are a few preliminary steps you have to take before you can use the folder feature. The first step is to create a subdirectory in your HOME directory for storing mail items. For example,

```
cd
mkdir mail
```

changes your working directory to $HOME and then creates a directory named *mail*. The next step is to edit your *.profile* file and add the following two lines.

```
folder=mail
export folder
```

This sets the shell environment variable *folder* to mail. Since the path of the directory does not begin with a slash (/), the directory is assumed to be in your HOME directory. You must re-execute your *.profile* for this new variable to take effect or log out and log back in to the system. By setting this variable you can use the *+filename* feature on the shell command line when you invoke **mailx**. When you use a plus (+) before a filename **mailx** knows to place the message list in the directory defined by the folder variable. To re-execute your *.profile* type . ./.profile and press Return. Next, edit your *.mailrc* file. Add the following line.

```
set folder=mail
```

This allows you to use a plus (+) inside **mailx** to have messages saved to files in the *$HOME/mail* directory.

Placing messages in folders You can relate a file to a folder. Each folder in the folder directory contains a set of messages. You can use any of the **mailx** commands that have filenames as arguments to store messages to a folder file. For example,

```
Save
```

saves the current message to a file with the name of the sender.

Multiple folders Once you have established different files for various types of messages you can use the folder command inside **mailx** to manipulate the messages stored within these different files. For example,

```
folder MBOX
```

sets the current mailbox to MBOX. Thus all subsequent commands are based on the assumption that MBOX is your new mail file (folder). From the shell command line you can use the *-f* option to have a certain file be the mailbox. This allows you to use the **mailx** commands on the messages stored in the specified file. For example,

```
mailx -f MBOX
```

causes the same setup as the internal *folder MBOX* command.

DIAGNOSTICS AND BUGS Because **mailx** is so complex, you may have some difficulties getting it to function exactly like you think it should. One suggestion: locate a **mailx** guru and buy the person off. It will be cheaper and much less frustrating. In perspective to other application type programs that exist in the world, **mailx** is not really that difficult to learn. It does have a few bugs worth mentioning though.

It currently does not support the full Internet addressing scheme. Someday when there is a stable standard for Internet it will do this.

If a line in the message only contains a . (period), there is a very high possibility that the entire message will not be delivered to a remote host. The period signifies "end of message" on the old mail program. The old mail program is the standard delivery program.

RELATED COMMANDS Refer to the **ed, ex, ls, pg,** and **vi** commands described in modules 39, 43, 84, 103, and 151.

You may wish to read about the **biff** (BSD), **notify** (SV4.0), and **vacation** commands in the User's Reference Manual which came with your system.

RELATED FILES The following list of files are used or updated by the **mailx** command.

$HOME/.mailrc	Your personal customization file read by **mailx** at startup.
$HOME/mbox	Your secondary *mailbox* for story messages.
/usr/mail/	The post office directory, where each user's *mailbox* is stored (SV).
/usr/spool/mail	The post office directory (BSD).
/usr/lib/mailx/mailx.help*	The files containing the help messages (SV).
/usr/lib/mailx/mailx.rc	The default customization file read by **mailx** at startup. Used by all users on the system (SV).
/tmp/R[emqsx]*	Temporary files used for various **mailx** requirements.

RETURN CODES The **mailx** command can be invoked with the -e option to only return a return code. If you have mail in your mailbox, **mailx** returns a 0 (zero); otherwise, **mailx** returns a nonzero value.

APPLICATIONS

You use the **mailx** command to exchange information with other users. The information may be any ASCII text; for example, memos, notes, program source, data, documents, or whatever information you need to send. The users may be local or remote depending on how your system is set up. If you have links to other systems via a LAN or modem, you can send mail to users on other systems.

TYPICAL OPERATION

In this activity you use the **mailx** (**mail** on BSD) command to send yourself a short message. Begin at the shell prompt.

BSD (Berkeley)

If you are on a BSD system use the /usr/ucb/mail command. You can use the **which** command to locate which **mail** command your PATH finds first. If it is /usr/ucb/mail, you can just type **mail**. Otherwise, change your PATH or type in the entire /usr/ucb/mail path. You may wish to set an alias to /usr/bin/mail. For example, place the following command in your .cshrc.

```
alias mail /usr/bin/mail.
```

---

1. Type **mailx mylogin** and press **Return**. Notice the Subject prompt appears.

```
cj> mailx mylogin
Subject: _
```

2. Now type in a brief subject line, **Sending mail to myself.** and press **Return**.

```
cj> mailx mylogin
Subject: Sending mail to myself.
_
```

3. Now you are in insert mode. You can type whatever message you wish. If you wish, use the following one. Remember to press **Ctrl-D** at the beginning of the line to terminate input.

```
cj> mailx mylogin
Subject: Sending mail to myself.

                                    April 22, 1989

Just thought I'd test the mailx command. So I'm sending myself a message.
UNIX is a powerful system of tools designed to increase productivity
among computer users.
^D
cj>
```

In this activity you invoke **mailx** to read any mail stored in your mailbox. Start at the shell prompt. You should wait a few minutes before continuing, since the message may still be en route to your mailbox. This is unlikely but on extremely slow systems it is possible.

1. Type **mailx** and press **Return**. Your screen should resemble the following display, although you may not have mail from other users.

```
cj> mailx
mailx version 2.2 9/9/88.    Type ? for help.
"/usr/mail/mylogin": 2 messages 2 unread
>O  1  bill      Tue Apr 12 08:31  5/134  "cabling problems"
 O  2  nancy     Tue Apr 12 10:03  4/98   "lunch in the break area"
 N  3  mylogin   Wed Apr 13 09:41  6/213  "Sending mail to myself."
```

2. Now you can read your new message. Just type **3** and press **Return**. Your message is displayed on your screen.

3. To delete the message type **d** and press **Return**. Since you are currently at message 3 no message list is needed. The d command deletes the current message.

4. Type **h** and press **Return** to display the header lines again. Notice the last message is gone.

5. Type **q** to quit **mailx**.

In this activity you send a file you created earlier to a user. Start at the shell prompt.

1. Type **mailx mylogin < first** and press **Return**. This sends the file *first* to yourself. The **mailx** command reads the standard input, thus allowing you to use redirection or pipe commands to send messages to specified users. A standard mail header is added to the beginning of the input file.

In this activity you create a .mailrc file to control how mailx function. Start at the shell prompt.

1. Type **cd** and press **Return** to return to your HOME directory.
2. Type **vi .mailrc** and press **Return**. This places you in the .mailrc file.
3. Now enter the following lines, pressing **Return** at the end of each line. When you have entered all lines, press **Escape** and type **ZZ** to save the file and exit vi.

```
cj> vi .mailrc
i
set LISTER="ls -xs"
set MBOX=$HOME/mail/MBOX
set PAGER="pg -cn -p '/usr/mail/mylogin'"
set cmd="lp -dlaser"
set hold
set EDITOR=vi
set folder=$HOME/mail
<Escape>ZZ
```

BSD (Berkeley)

```
cj> vi .mailrc
i
set LISTER=ls
set MBOX=$HOME/mail/MBOX
set PAGER=more
set cmd=lp
set hold
set EDITOR=vi
set folder=$HOME/mail
<Escape>ZZ
```

---

4. Since you set variables that require a directory named **mail** in your home directory, you need to create the directory. Type **mkdir mail** to create the mail directory.

The following activities will use some of the variables you set in the *.mailrc* file. In the following activity you use some tilde commands to help you generate a message.

1. Type **mailx mylogin** and press **Return**. Again we send mail to ourselves.
2. At the subject prompt type **Playing with the mailx command!** and press **Return**. Now you are ready to type in your message. You are in insert mode. Enter the following lines exactly as shown in the display.

```
cj> mailx nancy
Subject: Playing with the mailx command!
This is a brief message about tommorros big design meeting. I need to
know if I should order two dozen donuts or three?  PLeas let me know as
soon as possible. Thnaks

                                Myself
~e
```

3. You should now be in the **vi** editor with your message displayed. Now move around and correct the spelling mistakes in the message. The following list shows the motions and corrections; do not press Return after each line:

```
/mm<CR>
xfsiw<ESC>
jfL~
fsae<ESC>
jFnxp
ZZ
(continue)
```

   This will correct the spelling mistakes and save the message to a temporary file. You are placed back in insert mode of mailx.

4. To send the message press **Ctrl-D**.

```
(continue)
Ctrl-D
```

5. Type **mailx** and press **Return** to read the last two messages you have sent to yourself.
6. Press **Return** to read the first message or select the appropriate message number and press Return.
7. Press **Return** again to read the second message you sent.
8. Delete all messages from your mailbox by typing **d*** and pressing **Return**.
9. Type **q** and press **Return** to quit **mailx**.
10. Turn to Module 117 (SV), Module 72 (BSD) to continue the learning sequence.

Module 87
man (BSD)

DESCRIPTION

The external **man** command searches for the on-line copy of the UNIX Reference Manual entry for a particular command. If found, the manual entry is formated and displayed to your screen. The manuals are often refered to as "man pages." There is one man page entry per command.

NOTE

Some vendors no longer distribute System V Release 3.0 or later versions with on-line manual pages. Therefore, the **man** command does not exist. In most cases it has been replaced by the **help** facility. Which, unfortunately, seems to vary widely between vendors.

COMMAND FORMAT Following is the general format of the **man** command.

```
man  [ - ] [ -M path ] [ section ] title
man  [ - ] [ -M path ] [ -f file ]
man  [ - ] [ -M path ] [ -k keyword ]
```

System V

If the vendor supports it, it may vary from the following.

```
man [ -cdw ] [ -Tterm ] [ section ] titles
```

Options The following list describes the options and their arguments that may be used to control how **man** functions.

The formatted output from **man** is piped into the **more** command. The -s option of **more** is used to remove duplicate blank lines. Thus the formatted text is displayed to your terminal one page at a time. Refer to Module 90 for more information on the **more** command.

-M *path* — Provides a path of directories searched for manual subdirectories. Normally **man** searches the subdirectories in the /usr/man directory. If -M is used, **man** searches each of the directories in path for subdirectories that contain man entries. For example,

```
-M /usr/local:/share/man
```

causes **man** to search the /usr/local/man* and /share/man/man* directories for manual entries.

-k *keyword* — See the **apropos** command.

-f *file* — See the **whatis** command.

System V

-c	Causes **col** to be invoked. Invoked as a default. For more information about the **col** command refer to Module 21.
-d	Search current directory for entry. **man** defaults to the /usr/catman or /usr/man directory structure. Full man page entry name must be given (e.g. use **cat.1**, not **cat**).
-w	Display where the entry resides using the /usr/man path. The man entry is NOT formated and displayed. If -d is specified then the current directory is searched.
-T*term*	Format the entry for displaying on the given terminal type *term*. The default for *term* is 450. The value of the TERM variable is used as the output device if the -T option is not used.

Arguments The following list describes the arguments that may be passed to the **man** command.

section The section of the manual you want **man** to search. There are typically seven sections of the man pages. **man** searches all seven sections if you do not specify a section. You may specify a section before each title.

local	Format a local entry. Local entries are located in the /usr/man/manl directory (l as in local, not the number one). Each entry (file) in this directory must end with the .l suffix. You can add your own entries to this directory.
new	Search for and format a new man page entry.
old	Search for and format an old man page entry.
public	Search for and format a public man page entry.

title The name of the command you want **man** to search for and format to your screen. Multiple titles may be given. All titles should be entered in lowercase letters (considering all commands are supposed to be lowercase).

System V

local Usually not supported.

FURTHER DISCUSSION The on-line copy of the Reference Manual is not always available to you. If you are on a small system or your system was running low on disk storage, the system administrator may have removed some of all of the manual entries to gain disk space. The entries use up approxiatmately 3 megabytes of storage.

The following is a description of each section and the layout of the manuals in their on-line and physical format.

Sections There are twelve sections of the reference manuals. They are:

Section	*Description*
1	Commands and Application Programs
2	System Calls (used in programming languages)
3	Subroutines (used in programming languages)
4	File Formats
5	Miscellaneous
6	Games
7	Special Files
8	System Maintenance Procedures
local	Locally written man pages (third party software man pages.)
public	Public domain software man pages.
new	New software man pages.
old	Old/obsolete software man pages.

Some of the sections listed have subsections. The numbers of the sections follow the name of a command in parentheses. For example, cat(1) informs you that you can find the entry for the cat command in section 1 of the reference manual. The notation cu(1C) informs you that the cu command is used for communications. The following list describes the subsections.

Subsection	*Description*
1C	Section 1, Communications
1G	Section 1, Graphics
1M	Section 1, Maintenance (System Adminstration)
3C	Section 3, C programming language
3F	Section 3, Fortran programming language
3M	Section 3, Miscellaneous programming routines

Physical Layout There are three reference manuals containing only the entries relavent to the named audience. They are:

User's Reference Manual (Sections 1 & 6)
Programmer's Reference Manual (Sections 1, 2, 3, 4, & 5)
System Administrator's Reference Manual (Sections 1, 7, & 8 or 1m)

On-line Layout The on-line copy of the reference manual resides in these corresponding directories:

/usr/man/man1
/usr/man/man2
...
/usr/man/man8

System V

/usr/man/u_man/man[16]
/usr/man/p_man/man[12345]
/usr/man/a_man/man[178]

For more information on how to read a man page refer to the Introduction of a Reference Manual. Each vendor may implement the manuals as they desire. Most BSD type vendors support the BSD man page layout while most System V vendors that support the man pages tend to follow the AT&T standard.

DIAGNOSTICS AND BUGS The man pages that are already formated in the catman directory were formated for a type 37 terminal, therefore, they may not display correctly on your terminal. To reformat for your terminal use the -T option. If you want to send a man page to a typesetting device, you must again use the -T option and pipe your man command into the **lp (lpr)** command. It may be worth your time and effort to request assistance from your system administrator on this subject.

RELATED FILES The **man** command writes to the standard output. It searches the following directories:

/usr/man/man[1-8]/*
/usr/catman/man[1-8]/*
MANPATH=/usr/man:/usr/local/man:*your_man_directory*

System V

/usr/man/?_man/man[1-8]/*
/usr/catman/?_man/man[1-8]/*

APPLICATIONS

The **man** command is used to display the reference manuals for the current UNIX system. It provides on-line documentation for every standard command. As UNIX becomes more user friendly some of the systems have removed the **man** command and provided the **help** command.

TYPICAL OPERATION

In this activity you use the **man** command to generate the man page for the **pwd** command. Begin at the shell prompt.

1. Type **man pwd** and press **Return**. The man page for pwd appears after it is formated by the nroff command.
2. Turn to Module 5 (BSD) to continue the learning sequence.

Module 88
mesg

DESCRIPTION

The external **mesg** command allows or denies other users the privilege of sending messages to your terminal screen. By using the **mesg** command you can prevent people from **write**ing or **talk**ing to you while you are logged on the system.

COMMAND FORMAT The general format of the **mesg** command follows.

```
mesg [ -n | -y ]
```

Options The following list describes the options and their arguments that may be used to control how **mesg** functions.

-n	Prevent other users from sending messages to your terminal.
-y	Allow other users to send messages to your terminal.

If no options are given the current message mode is returned.

FURTHER DISCUSSION If you keep getting interrupted by other users, enter

```
cj> mesg n
cj> mesg                # use mesg alone to check current state
```

and you can continue on in peace. The **mesg** command only works for the current login session. If you log out and back in, you must reset the **mesg** mode as desired. The default is to allow other users to send messages to you.

The **mesg** command prevents other users from writing to your terminal by changing the write permission for others on your /dev/ttyXX terminal port.

RELATED COMMANDS Refer to the **write** and **talk** commands described in modules 161 and 132, respectively.

RELATED FILES All /dev/tty* terminal files are valid for message mode control.

RETURN CODES The **mesg** command returns the following codes:

0	messages are allowed
1	messages are denied
2	**mesg** failed with an error

APPLICATIONS

The **mesg** command is used to control when people can and cannot interrupt your work by writing to your terminal screen. It is automatically turned on when you are in **vi**. This prevents other users from garbling your edit screen. It is a desirable feature if someone keeps bothering you and you are starting to become annoyed.

TYPICAL OPERATION

In this activity you use the **mesg** command to deny other users write permission to your terminal, then you set the mesg mode back. Begin at the shell prompt.

1. Type **mesg** and press **Return** to display the current message mode. It should be set to *yes*.
2. Type **mesg -n** and press **Return** to change the mode to *no*. Now, if feasible, contact a fellow user and have him/her try to **write** to you. The person should receive a message stating "Permission denied." Or, you can try to write to your own login.
3. Type **mesg -y** and press **Return** to change the mode back to *yes*. If you like, have the user try to **write** you a message now. You should receive each line of the message as the user presses **Return** at the end of each line.
4. Turn to Module 132 to continue the learning sequence.

Module 89
mkdir

DESCRIPTION

The external **mkdir** command makes a new directory. You can pass multiple arguments to **mkdir**. Each argument is used by **mkdir** as the name of a new directory to create.

COMMAND FORMAT Following is the general format of the **mkdir** command.

```
mkdir [ -m mode ] [ -p ] directory_list
```

BSD (Berkeley)

```
mkdir directory_list
```

Options The following options and their arguments may be used to control how **mkdir** functions.

-m *mode* — Mode to use for new directories. This allows you to specify what mode all of the new directories will have when they are created. The default is 777 minus your **umask** number or the system's default umask number.

-p — Parent directory creator. If you specify a pathname to create a new directory and the parent directories do not exist, **mkdir** will create them as needed.

BSD (Berkeley)

The BSD version does not support any options.

Arguments The following arguments may be passed to the **mkdir** command.

directory_list — The list of names to be created as directories. The directory names may be pathnames or single filenames.

FURTHER DISCUSSIONS Each new directory created will have two entries placed in it by **mkdir**. The first entry is . (dot) which represents the directory itself (current directory). The second entry, .. (dot dot), represents the parent directory of the new directory.

NOTE

You must have write permission for the directory where you are creating the new directories. If you do not have write permission, you will not be allowed to create the new directories.

When you create a new directory the system builds a new inode (information node) for the directory. This allows access to the data stored in the directory file and informs the system that the file is a directory type file. The directory file is a preformatted file, containing the filename and its related inode number.

The permissions given to a new directory are set to your current mode. When you initially begin to log in, your mode is set to 777 (-rwxrwxrwx). If the system's profile or your .profile performs an **umask**, your modes may be different. Use the **umask** command to check your current modes. Type umask and press Return to display your current umask value. Refer to the module on **chmod** for an explanation of the modes.

The user-ID and group-ID of the new directories are the same as the real user-ID and group-ID of the creating process. If you create a directory, then it will be owned by you and have your group-ID.

DIAGNOSTICS AND BUGS If an error occurs while creating a directory, **mkdir** writes a diagnostic message to the standard error and returns a nonzero exit status upon exiting.

RELATED COMMANDS Refer to the **rm**, **rmdir**, **ksh**, and **umask** command described in modules 114, 115, 71, and 147, respectively.

APPLICATIONS

You use the **mkdir** command to create new directories, thus building a hierarchy of directories to maintain your files in an orderly fashion.

It may be useful for you to consider setting a standard naming convention or structure for your directory hierarchy. Try to plan ahead when creating directories. The most common mistake made by inexperienced UNIX users is the lack of subdirectories. It is much easier to locate a file if you have your files categorized in subdirectories.

The rule of thumb for the number of files in a directory is 50 to 100. These numbers are based on performance, not on limits. The more files in a directory the longer it takes to retrieve the information from disk. Therefore, it is advisable to only have 50 files per directory. If you have hundreds of files to store, you probably want to create subdirectories with the same basic name and an incremental extension. This allows you to place 50 to 100 files in each of the directories, while maintaining a consistent name for the directories.

The **ls -l** and **ls -F** command can be used to show if an entry in a directory is a subdirectory.

TYPICAL OPERATION

In this activity you use the **mkdir** command to create a new subdirectory. Begin at the shell prompt.

1. Type **mkdir misc misc/misc** and press **Return** to create a subdirectory called *misc* which has a subdirectory named *misc*.
2. Type **ls -Ral misc** and press **Return** to display the misc and misc/misc directory entries in long format. Your screen will be similar to this:

```
cj> ls -Ral misc
total 6
drwxr--r-- 3  mylogin     ts     512 Feb  4 08:33 .
drwxr--r-- 13 mylogin     ts     512 Jan 24 06:43 ..
drwxr--r-- 2  mylogin     ts      24 Feb  4 08:33 misc
```

```
misc/misc:
total 4
drwxr--r-- 2  mylogin     ts      24 Feb  4 08:33 .
drwxr--r-- 3  mylogin     ts     512 Feb  4 08:33 ..
```

Your listing will differ in link numbers, sizes, dates, and times.

3. Turn to Module 115 to continue the learning sequence.

Module 90
more/page

DESCRIPTION

The external **more** command allows you to peruse a file on your terminal. It is a filter that reads the standard input or files. It lets you view one screen of text at a time.

The **page** command performs the same function as **more** with a different screen control. The screen is cleared before each full screen of text is displayed. It also displays ROWS - 1 rather than ROWS - 2 rows of text, where ROWS is the number of lines on your terminal screen.

COMMAND FORMAT Following is the general format of the **more** command.

```
more [ -cdflsuw ] [ -rows ] [ +linenum ] [ +/pattern ] [ file_list ]
page [ -cdflsuw ] [ -rows ] [ +linenum ] [ +/pattern ] [ file_list ]
```

BSD (Berkeley)

```
more [ -cdflsu ] [ -rows ] [ +linenum ] [ +/pattern ] [ file_list ]
page [ -cdflsu ] [ -rows ] [ +linenum ] [ +/pattern ] [ file_list ]
```

Options The following list describes the options and their arguments that may be used to control how **more** functions.

-c Draw each screen of text line-by-line. Each line is erased just before the new text is written to the line. This provides a more controlled screen, allowing for easier reading of existing text while new text is being displayed. If the terminal cannot perform a clear-to-end-of-line function, the option is ignored.

-d Changes the prompt to be "Press space to continue, 'q' to quit." from "--More--(9%)." If an incorrect response is entered, then the message "Press 'h' for instructions." is displayed instead of sounding the terminal bell. This is useful for application shell programming where **more** is used as an interface to unsophisticated users.

-f Do not fold long lines. Count logical lines instead of screen lines. Normally **more** truncates lines longer than the width of the screen (environment variable COLUMNS). This option allows you to pipe **nroff** output to **ul** then into **more** without having lines folded (truncated) because of embedded escape sequences.

-l Do not process form feeds as special characters. Normally **more** pauses before each form feed, assuming the end of a page of text has been reached. A form feed at the beginning of a file normally causes **more** to clear the screen before displaying text from the file.

-s — Squeeze out adjacent blank lines. Two or more blank lines grouped together are reduced to one blank line. This increases the amount of text displayed on the screen. It is very useful when viewing **nroff** output.

-u — Suppress underlining and standout (bold) mode. The normal sequences for underlining and standout mode are stripped from the output.

-w — Prompts and waits for any key to be pressed before exiting. By default *more/page* will exit without waiting upon reaching the end-of-file.

-rows — Specifies the number of rows (lines) to display on your terminal screen. Resets the size, in lines, of your window to *rows*

+linenum — Begin displaying text at line number *linenum* of the input.

+/pattern — Begin displaying text two lines before the line containing the regular expression *pattern*.

Arguments The following list describes the arguments that may be passed to the **more** command.

file_list — The list of files to be read by **more**.

If no files are specified, more reads from the standard input, which may be your keyboard or from a pipe.

Internal Commands After **more** starts displaying text you can enter commands at the "--More--" prompt to perform certain functions. The following commands may be used.

*n*Space — Display *n* more lines of text. If *n* is not specified, then 22 more lines of text (one screen) are displayed.

= — Display the current line number.

' (single quote) — Go to the line where the last search began. If a search has not been performed in the current file, then go to the beginning of the file. Only works when reading files. Does not work on standard input.

. (period) — Repeat the previous command.

n/expr — Search for the *n*th occurrence of the regular expression *expr*. If the input is a file rather than standard input (a pipe) and there are fewer than *n* occurrences of *expr*, the position within the file remains unchanged. If *n* occurrences of *expr* exist, then **more** displays two lines before the location of *expr*. If you backspace over the / the search is canceled. While typing the *expr* you may use the Backspace and Kill characters.

*n*Ctrl-B
*n*b — Skip *n* full screens back and display the next full screen of text. Does not work on standard input if the amount of text is very large.

*n*Ctrl-D
*n*D
*n*d — Scroll 11 more lines of text onto the screen. If *n* is specified the scroll size is changed from 11 to *n* lines.

*n*f — Skip *n* full screens ahead and display the next full screen of text.

h — Help. Display a description of all of these help commands.

*n*n — Search for the *n*th occurrence of the last regular expression you specified.

Q
q — Quit the program (exit **more** or **page**).

*n*s — Skip *n* lines forward in the text and display the next full screen of text.

v — Start the **vi** editor at the current line. Only works when viewing files on most systems. Does not work on standard input.

*n*z — Reset the window size to *n* lines. The z performs the same function as pressing Spacebar, except if *n* is specified the window size is reset.

:f — Display the current filename and line number.

n:n — Skip to the *n*th next file given on the command line. If no more files exist **more** exits. If fewer than *n* files exist then **more** skips forward to the last file. If **more** is reading the standard input, then the command is ignored. The following line is displayed at the bottom of your screen.

```
--More--(Next file: filename)
```

n:p — Skip to the *n*th previous file given on the command line. If *n* is greater than the number of previous files, then **more** returns to file one. If you have displayed part of the current file, **more** returns to the top of the current file. If **more** is reading the standard input, then the command is ignored. The following line is displayed at the bottom of your screen.

```
--More--(Next file: filename)
```

:q
:Q — Quit for **more** or **page**.

!*command* — Invoke a shell which executes *command*. For example,

```
!lp -dlj %
```

performs a **sh -c lp -dlj filename** where filename is the current file you are displaying. The percent (%) sign is expanded to the current filename. The exclamation point (!) is expanded to the last command specified. If there is a current filename, the % is not expanded. You can escape the % and ! by typing \% and \!.

SCREEN VIEWING After each screen **more** pauses and displays the "--More--" prompt at the bottom left corner of the screen. If the input is from a file, instead of a pipe (standard input), **more** reports the percentage of total text displayed so far and the prompt becomes "--More--(17%)." If you press Spacebar at the prompt, the next screen of text is displayed. Pressing Return displays the next line of text.

SCREEN CONTROL The screen control characteristics used by **more** are retrieved from the /etc/termcap database. Some System V UNIX systems implement **more**; in this case the characteristics may be retrieved from the terminfo database. The clear screen sequence is needed, along with the default window size and cursor positioning. If the window size is 24 lines, **more** displays 22 lines of text.

OUTPUT STYLE If the standard output is not a tty (terminal), **more** behaves like the **head** command. It displays a header and the contents of the file for each file listed.

ENVIRONMENT VARIABLE CONTROL You can set the environment variable MORE to any flags you desire to use each time you invoke **more**. For example, if you always wish to use the -d option, you perform the **ksh (sh)** command,

```
MORE='-d'; export MORE
```

or the **csh** command,

```
setenv MORE -d
```

to cause all executions of **more** to use the -d option. Normally, you place these commands in your .cshrc or .profile (.kshrc) files appropriately.

DIAGNOSTICS AND BUGS Skipping backward may be extremely slow on large files.

RELATED COMMANDS Refer to the **cat** and **pg** commands described in modules 13 and 103.

RELATED FILES The **more** command uses the following files.

/usr/share/lib/termcap Terminal capabilities database.
/usr/lib/more.help The help file — for more.

BSD (Berkeley)

/etc/termcap	Terminal capabilities database.
/usr/lib/more.help	The help file for **more**.

APPLICATIONS

The most common use of **more** is to display large files or output from a pipe to your screen one screen at a time. It is also useful for help screens in shell applications. You can create a help file exactly a multiple of 22 lines and have **more** display it within your shell script. This provides you with a professional and consistent interface for your users.

TYPICAL OPERATION

In this activity you use the **more** command to display the contents of three files. Begin at the shell prompt.

1. Type **more /etc/passwd /etc/group $HOME/file1** and press **Return**. The first screen of the passwd file will appear.
2. Now press **Spacebar**. The next screen of data is displayed (if the file has more than 22 lines of user accounts. If the file does not, the next file is started.)

3. Now type **2:n** to skip to the /etc/group file. Your screen looks like this:

```
...Skipping
...Skipping to file /u1/ts/mylogin/file1

--More--(Next file: /u1/ts/mylogin/file1)
```

4. To return to the second file type **:p** to skip backward to the /etc/group file.

```
...Skipping
...Skipping to file /etc/group

--More--(Next file: /etc/group)
```

5. Press **Spacebar** to move forward one screen.
6. Press **Spacebar** until the next file ($HOME/file1) appears.
7. Type **!lp %** and press **Return** to print a copy of your $HOME/file1 file to the default printer.

BSD (Berkeley)

Type **!lpr %** and press **Return** to print a copy of your $HOME/file1 to the default printer.

---

8. Type **q** to exit from **more**.
9. Turn to Module 97 to continue the learning sequence.

Module 91
mv

DESCRIPTION

The external **mv** command moves a file from an existing location to a new location. It is also used as the *rename* command to change the name of a file. It has three formats allowing you to:

- Rename a file with a new filename.
- Move one or more files to reside under a different directory.
- Rename a directory with a new directory name.

COMMAND FORMAT The general formats of the *mv* command follow.

mv [**-if**] [**-**] *filename new_filename*
mv [**-if**] [**-**] *filename directory*
mv [**-if**] [**-**] *old_directory new_directory*

Options Only one option may be used to control how **mv** functions.

-f — Force the move to occur. The response of "yes" is assumed and the move is performed. No interactive response is requested.

If the standard input is not the terminal keyboard, then the -f option is assumed.

-i — The command runs in interactive mode. If the destination file being moved to already exists, **mv** prompts you with the filename being moved followed by a question mark. If you respond with any string beginning with a **y**, the move is performed. Any other response causes **mv** to skip to the next move.

- — Interpret all arguments that follow as filenames. This allows you to specify filenames and directories that begin with a hyphen (-). This can become helpful if a filename is created that begins with a hyphen.

Arguments following list describes the arguments that may be passed to the **mv** command.

filename — The existing file being moved or renamed.
filename_list — The names of files to be moved to a new destination. The destination must be a directory.
new_filename — The new filename the existing file will be given.
directory — The name of the directory the files will reside in after the *mv* is performed.
old_directory — The existing directory being renamed.
new_directory — The name of a newly created directory. The destination directory.

FURTHER DISCUSSION The first format renames a file. The name of the existing file is changed to the name of new_filename. The new_filename and the existing filename cannot be the same full pathname. They may have the same filename, providing they reside in different directories. If the new filename is an existing file and you have write permissions, its contents are

lost. If the destination is a link to another file, the link is lost and the destination filename points to the new file.

The second format moves one or more files to a different directory. The filename_list is a list of pathnames being moved. The basename of each pathname is used for the destination filename in the specified directory. For example, if you wanted to move all files beginning with memo to the memodir directory, you would type mv memo* memodir.

The third format renames a directory. The name of the directory is changed to the new directory name you specify. The same restrictions apply to directory renaming as file renaming. If you are renaming directories, the two directory names reside in the same file system.

The **mv** command actually performs an **ln** and an **rm**. A link (**ln**) is created from the existing file to the new filename. Then the existing filename is removed (**rm**). If the new filename already exists, it is removed (**rm**) before the move attempts to link.

Before **mv** performs these steps it checks the destination file for write permissions. If you do not have write permissions on the destination file or directory, **mv** will display the mode (**chmod**; Module 17) and request a response from standard input. If the input line you type starts with a "y," **mv** attempts to move the file to the target. If the ownership permissions allow you to write to the target, the move will occur. If you do not have permission to write to the destination, the **mv** will fail.

If you are moving files across file systems, the **mv** command performs a **cp** and then an **rm** command. The file is copied to its destination and then the existing file is removed. The new file is owned by you if you moved it.

Some common formats are:

```
mv letter* LTR        # move all files beginning with letter to the
                      # LTR directory
mv pup dog            # rename the file "pup" to "dog"
mv LTR LTRS           # rename directory "LTR" to "LTRS"
```

DIAGNOSTICS AND BUGS The **mv** command must copy the existing file to the destination and remove the original if the files reside on different file systems. All links are lost if this action occurs.

RELATED COMMANDS Refer to the **cp**, **ln**, **rm**, **chmod**, **chown**, and **ls** commands described in modules 24, 76, 114, 17, 18, and 84, respectively.

APPLICATIONS

The **mv** command is used whenever you need to rename or move a file. It allows you to organize your files as you like. For example, if you had written 15 letters with names of LTR1 through LTR15, you use **mv** to move them to a subdirectory named LETTERS. The command mv LTR[0-9]* LETTERS would move all of your letters to the LTRS directory.

The **mv** command is often used to rename a directory. To rename your LETTERS directory to LTRS you type mv LETTERS LTRS.

Another common use is to rename default filenames with a meaningful filename. For example, the C compiler creates the a.out executable file. You use can **mv** to rename a.out to a new filename so you know what the program does; for example, mv a.out chess.

TYPICAL OPERATION

In this activity you use the **mv** command to rename a file. Begin at the shell prompt.

1. List your HOME directory by typing **ls** and pressing **Return**. Notice the file named *file1* in the following display.

```
cj> ls -x
bin     calendar  db   file1     file2     letters
```

2. Rename file2 to stuff by typing **mv file2 stuff** and pressing **Return**.
3. List the directory to verify your filename changed by typing **ls** and pressing **Return**. The following display reflects the change:

```
cj> ls -x
bin     calendar  db   file1     letters   stuff
```

4. Turn to Module 76 to continue the learning sequence.

Module 92
newgrp (SV)

DESCRIPTION

The internal **newgrp** command lets you change to a new group. Groups allow users belonging to the same group to access directories and files based on group permissions. To change groups you must be listed in the /etc/group file as a user in the desired group. When you execute **newgrp**, your shell always **execs** a new shell for you; thus your environment is always reset.

BSD (Berkeley)

The approach to group access and group membership under BSD systems is different from that of System V. If you are listed in multiple groups in the /etc/group file, you are a member of each group at all times. Thus there is no need for a **newgrp** command.

NOTE

Some vendors provide the newgrp and support the BSD idea of groups. The system administrator can configure the system to be SV like or BSD like.

COMMAND FORMAT Following is the general format of the **newgrp** command.

```
newgrp [ - ] [ group ]
```

Options The following option may be used to control how **newgrp** functions.

The environment is changed to reflect an initial login environment for your user name with the new group identification. Has the same effect as logging in to the system but with a new group ID.

Arguments The following argument may be passed to the **newgrp** command.

group Changes your group ID to the ID of the group you specified.

If no argument is given, your group ID is changed to the group specified in /etc/passwd for your user name. It has the same basic effect as logging out and logging back in with your normal group ID.

FURTHER DISCUSSION If the requested group exists and you are allowed access to it, your group ID will be changed. If you are not allowed access to the requested group or the group does not exist, your environment is reset to the exported environment before the **newgrp** was performed.

The current environment (all exported variables) is passed to the new shell when **newgrp** is executed. Variables that have not been exported are lost. System variables are reset to default values if they were not exported. The **export** command is described in Module 47.

To know which variables have been exported use the **env** or printenv command. The **env** command is described in Module 41, the **printenv** command is described in Module 107. The **set** command will list all variables that have been set. It lists both exported and nonexported variables.

If you need to be a member of a group other than the one you are in, contact your system administrator. The administrator can add your logname to the /etc/group file for the new group.

If the group has a password and you are not listed in the /etc/group file as a group member, you will be requested to enter the group password. If you enter the correct password, a new shell will be **execed** and your new shell will assume the new group ID. The following section describes problems with group passwords.

DIAGNOSTICS AND BUGS Because group passwords promote poor security practices and there is no good way to implement them, it is advisable not to use them.

You may receive the following responses from newgrp depending on the circumstances described:

unknown group	The group you requested does not exist in the /etc/group file.
Sorry	You are NOT allowed access to that group. This means your user name is not listed as a member of the group in the /etc/group file.

RELATED COMMANDS Refer to the **login** command described in Module 77 and the **ksh** command described in Module 71.

RELATED FILES The following list describes the files that **newgrp** uses.

/etc/passwd	The system's passwd file, used to reset your group ID back to your default login group ID.
/etc/group	The system's group file, used to locate valid groups and access permissions for your new group requests.

APPLICATION

Newgrp is useful in switching between groups to keep programming projects and other business groups separate. A manager may be in multiple groups while each of the manager's employees is only in one group. This allows the manager to access the employee's files even though they belong to different groups.

TYPICAL OPERATION

NOTE

System V provides an environment for being in one group at a time, whereas Berkeley allows you to be in multiple groups simultaneously. Therefore, Berkeley does not provide a newgrp command since there is no need for one.

1. Type **ps** and press **Return** to display the process ID (PID) of your current shell. Remember this number so you can compare it to your new shell after you execute **newgrp**.

2. Type **newgrp dvlp** and press **Return**. After a few seconds delay you will see your prompt reappear. You will need to locate a valid group you can change groups to using **newgrp**. You might ask your system administrator to add you to a new group if you have a good reason. Some sites only implement one user group.

```
cj> newgrp dvlp
cj>
```

   You should now be able to access files and directories set with group permissions that you may not have previously been able to access.

3. Type **ps** and press **Return** to display your new shell's process ID. Notice the process IDs are different. This is because a new process was started for your new shell.

4. Type **newgrp xxx** and press **Return**. If you request a group that does not exist in the /etc/group file, you receive the following message and remain in your current group, although you will be in a new shell and possibly a reset environment.

```
cj> newgrp xxx
newgrp: Unknown group
```

5. Type **newgrp root** and press **Return**. If you request a group that you are not allowed to be a member of, you receive the following message and remain in your current group.

```
cj> newgrp root
newgrp: Sorry
```

6. Turn to Module 100 to continue the learning sequence.

Module 93
news (SV)

DESCRIPTION

The external **news** command informs the user of any late breaking news. The news items are usually related to the system. The editor is most likely to be the system administrator. Depending on your system's setup and operations you may be able to post news for others to view.

COMMAND FORMAT Following is the general format of the **news** command.

```
news [ -ans ] [ items ]
```

Options The following list describes the options and their arguments that may be used to control how **news** functions.

-a	All news items are displayed in their entirety regardless of modification times. The .news_time file is not modified.
-n	All filenames (names of the current items) are displayed regardless of the modification time. The .news_time file is not modified.
-s	Displays the total number of news items without displaying item names or contents. The .news_time file is not modified. This option is often used in the user's .profile file or the system's /etc/profile file to inform you of how many news items exist.

Arguments The following describes the argument that may be passed to the **news** command.

items	A list of filenames stored in /usr/news that contain news type information.

FURTHER DISCUSSION Each news item is stored in a separate file. All news files are stored in the /usr/news directory. When you execute **news**, it checks a file named .news_time in your $HOME directory for the last time you viewed the news items. Based on the modification date of this file, **news** will print all items that have been posted since the specified date and time.

If you press Delete while **news** is displaying a news item, **news** stops displaying the current item and starts displaying the next item. If you press two Deletes within a one-second time frame, **news** terminates.

RELATED FILES The following list of files are used by the **news** command.

/vars/news/*	Items stored for release and viewing
/$HOME/.news_time	The modification time is used to know when the user last read news items.
/etc/profile	Often contains the "news -s" command, which shows you how many news items are currently posted for reading.
standard out	The output is written to standard output.

APPLICATIONS

The **news** command is used to keep users up to date on system activities, corporate events, extremely important sports events (World Series, Superbowl, etc.), and other pertinent news items. News is often used to broadcast warnings or software release notes. The type of news and who can post it is decided by the system administrator or corporate policy.

TYPICAL OPERATION

In this activity you use the **news** command to see if any news items have been posted on your system. If so you use **news** to read them. Begin at the shell prompt.

1. Type **news -s** and press **Return**. The number of news items posted on your system are returned. If a number is not returned, then no items are currently posted. If your system has news items posted, your screen should resemble the following display.

   ```
   cj> news -s
   news: 2
   ```

   The above display shows 2 news items are available for viewing. If nothing is displayed, no news items are available for viewing. You may want to create a news item just for testing. Simply type **vi /usr/news/test** and press **Return**. Type in a brief message, save the file, and exit vi. Retype the **news -s** command as previously discussed.

2. Type **news** and press **Return** to display the two news items.
3. Turn to Module 156 to continue the learning sequence.

Module 94
nice

DESCRIPTION

The external **nice** command executes a specified command at a lower CPU priority than normal. Thus its name is derived from the fact the command is being nice to the rest of the users. The lower your priority the slower your program runs and the longer it takes to complete.

The super-user can increase the priority of a command by using **nice**. Instead of specifying *-n*, a *--n* is given.

The nice value of a process can be viewed using the **ps -l** command. The NI column displays the **nice** value. The higher the value the lower the CPU priority (the slower the process executes).

COMMAND FORMAT Following is the general format of the **nice** command.

```
nice [ -n ] command [ options ] [ arguments ]
```

Options The following option may be used to control how **nice** functions.

-n Decreases the scheduling priority by the value of *n*. The value of *n* can be from 1 to 19.

If you do not specify a number, 10 is used.

Arguments The following arguments may be passed to the **nice** command.

command The name of any command you normally execute. This command's process ID is assigned a lower priority level.

options The options to be used by the *command*.

arguments The arguments to be used by the *command*.

C shell

The C shell uses a + to lower your CPU priority and a – to increase it. The **nice** command is an internal command.

DIAGNOSTICS AND BUGS If you specify a number greater than 19, 19 is used.

RETURN CODES The exit status of *command* is returned.

APPLICATIONS

The **nice** command is beneficial in controlling the use of the CPU. If you are running large jobs, such as document formatting or other various CPU hogs, you should use **nice**. These CPU and/or IO intense jobs slow down the interactive jobs unless you lower their priority. This allows you to process large CPU intensive jobs while continuing to perform interactive tasks without as much delay.

TYPICAL OPERATION

In this activity you use the **nice** command to lower the priority of a command. Then you check the priority by using the **ps** command. Begin at the shell prompt.

1. Type **nice find / -name core -print > /dev/null &** and press **Return**. This places the find command in background at a lower priority.
2. To check the priority of the **find** command type **ps -l** and press **Return**.

```
cj> ps -l
F S   UID   PID  PPID  C PRI NI   ADDR   SZ     WCHAN TTY      TIME COMD

1 R    60  4947 4771 14 181 20 bedc81  820            tty3p3   0:00 ps
1 S    60  4771 1     0 168 20 a02ea0  500   1abd6a0 tty3p3   0:02 ksh
1 S    60  4945 4771  3 148 30 594ab0  392    897380 tty3p3   0:03 find
```

   The NI field is the NICE value for each process. The higher the value the nicer the command.

3. Type **kill 4945** and press **Return** to kill the **find** process. Replace 4945 with the PID number for your find process.
4. Turn to Module 96 to continue the learning sequence.

Module 95
nl (SV)

DESCRIPTION

The external **nl** command is used to precede input text with line numbers. The input may be a file or the standard input. Special delimiters may be used in the input text to control how **nl** numbers the lines.

The **nl** command views the input text as logical pages. There are three possible sections within each logical page. A section of a logical page may be empty. Each section may be numbered differently. The following table lists the strings that separate the logical pages into sections. These strings must be placed in the input data. The sections and their delimiter strings are:

Section	*Input Delimiter*	*Meaning*
header	\:\:\:	Marks the beginning of the header section.
body	\:\:	Marks the beginning of the body section.
footer	\:	Marks the beginning of the footer section.

If no delimiters exist in the input, **nl** performs line numbering on the entire file as if it were one logical page consisting of only a body section.

To illustrate the logical page concept let us assume we have the following file:

```
cj> cat afile
\:\:\:
The header section
\:\:
The body section of the file.
\:
The footer section
```

This file makes up one logical page for **nl** to process. The \:\:\: string informs **nl** that the header section follows. The \:\: string precedes the main body section of the file. And last, the \: sets apart the footer section. Each logical page may contain one or more of these sections. If no section delimiters are found by **nl**, then all input is considered to be the body section.

COMMAND FORMAT Following is the general format of the **nl** command.

```
nl [ -btype ] [ -ddelim ] [ -ftype ] [ -htype ] [ -iincr ] [ -lnum ] \
   [ -nformat ] [ -p ] [ -ssep ] [ -vstart ] [ -wwidth ] [ file ]
```

Options The following list describes the options and their arguments that may be used to control how **nl** functions.

-b*type* Specifies which lines in the *body* section are to be numbered based on the given *type*. The default type of numbering is "t." The available types are:

Type	*Function*
a	Number all lines.
n	Do not number the lines.
p*string*	Number only the lines containing the regular expression *string*.
t	Number only lines containing ASCII text that can be displayed on your terminal screen.

-d*delim* Changes the delimiter characters used to break the logical page into sections. The normal characters are \:. You must specify two characters. If you do not, the colon (:) remains as the second default character. No space is allowed between the -d and *delim* string.

-f*type* Specifies which lines in the *footer* section are to be numbered based on the given *type*. The default type of numbering is "n." The available types are listed under option -b.

-h*type* Specifies which lines in the *header* section are to be numbered based on the given *type*. The default type of numbering is "n." The available types are listed under option -b.

-i*incr* The number *incr* is used to increment the line numbers. Normally, the line numbers are incremented by 1.

-l*n* Considers *n* blank input lines as one output line. Numbers every *n*th blank line in a group of adjacent blank lines. The -ha, - ba, and/or -fa options must be specified.

-n*format* Specifies the format used to display the line numbers. The possible values are:

Format	*Function*
ln	Left justify with leading zeroes suppressed.
rn	Right justify with leading zeroes suppressed.
rz	Right justify with leading zeroes.

If no *format* is specified *rn* is assumed.

-p The line numbers are not reset for each new logical page. Normally each logical page (not section) begins with line number one.

-s*sep* Use string *sep* to separate the line numbers from the text. The default is a tab.

-v*start* Use the value of *start* to begin line numbers. Normally, line numbers start at 1.

-w*width* The field width for the line numbers is set to *width* positions. If no value is specified 6 is used.

Arguments The following argument may be passed to the **nl** command.

file One file of input data. The **nl** command can only read from one source of input.

If no file is specified, **nl** reads from the standard input. Therefore, it can be used in pipes.

RELATED COMMANDS Refer to the **pr** command described in Module 105.

APPLICATIONS

The **nl** command is designed to provide line numbering for files. It can provide various line numbering methods and is fairly simple to use. One of the most popular uses is numbering lines beginning at a certain number and incrementing the line numbers by a multiple of 10 or 100. It is often used to number the lines on a program listing before sending the listing to the printer.

TYPICAL OPERATION

In this activity you use the **nl** command to place line numbers before each file listed in the /bin directory. Begin at the shell prompt.

1. Type **ls /bin | nl -nrz -w3 -s" "** and press **Return**. This displays the contents of the /bin directory, placing a 3-character, zero-filled line number before each filename.
2. Turn to Module 21 to continue the learning sequence.

Module 96
nohup

DESCRIPTION

The external **nohup** command executes a UNIX command immune to the hang-up and quit signals; thus its name **no hang-up.** The primary objective of the **nohup** command is to have a process continue executing if you get logged off the system because of a terminal line hang-up.

C shell

The **nohup** command in C shell is a built-in command. This internal command does not redirect output to the nohup.out file like its System V counterpart.

Any command ending with an &, which places the process in background mode, is automatically immune to hang-up signals.

COMMAND FORMAT Following is the general format of the **nohup** command.

```
nohup command [ options arguments ]
```

Arguments The following arguments may be passed to the **nohup** command.

command	The command you wish to execute which will not terminate if a hang-up or quit signal is received.
options	The options for the *command*
arguments	The arguments for the *command*

FURTHER DISCUSSION The **nohup** command allows you to start a job and then log off the system. The job continues running even though a hang-up signal was sent to it. The standard output and standard error of **nohup** and your command are redirected to the file nohup.out. If **nohup** cannot create nohup.out, then it tries to write to $HOME/nohup.out. You can redirect the output by specifying the output file. For example,

```
cj> nohup acmd 2>&1 > acmd.out &
```

executes the *acmd* in background, ignoring hang-up and quit signals. The output is sent to the acmd.out file.

If you want the **nohup** command to apply to several commands, you must write a shell script containing the desired commands. For example,

```
cj> cat sortdata
for FILE in data1 data2 data3 data4
 do
   sort $FILE -o $FILE
 done
```

The **sortdata** shell script sorts each file named *data1* through *data4*. To run this script in background without terminating when you log off the system, you type

```
cj> nohup sortdata &
```

You must be careful in applying the **nohup** command to commands on the shell command line. For example,

```
cj> nohup command1; command2
```

nohups the first command but not the second.

If the following command is specified on the command line, the shell will complain about the syntax. You cannot specify subshell process as another command's arguments.

```
cj> nohup ( command1 )
```

APPLICATIONS

You usually use the **nohup** command to start long jobs and then log off the system. It can be used for jobs you want to execute and make certain they are not terminated if you are logged off the system or if a quit signal is received.

TYPICAL OPERATION

In this activity you use the **nohup** command to start a job in background, log off the system, log back in, and check on the process. Begin at the shell prompt.

1. Type **nohup spell /etc/passwd /etc/passwd /etc/passwd | wc > spchck &** and press **Return**.
2. Immediately press **Ctrl-D** to log off the system.
3. Now log in to the system. Type your login name **mylogin** and press **Return**.
4. When the system prompts you, type your password and press **Return**.
5. As soon as you are logged back in type **ps -u mylogin** and press **Return**. Depending on the length of your passwd file and the speed of your system the **spell** command may still be executing. If it is not executing, check the output. If it is, wait for it to complete. You will have to check using **ps** periodically.
6. Type **cat spchck** and press **Return**. There should be a line with the number of misspelled words that were found in the /etc/passwd file.
7. Type **rm spchck** and press **Return**.
8. Turn to Module 109 to continue the learning sequence.

Module 97
od

DESCRIPTION

The external **od** command provides a way to view the contents of a file byte-by-byte or word-by-word. A byte is eight bits and a word is two bytes (16 bits). The format of each byte or word can be octal, decimal, hexadecimal, or ASCII. You can use **od** to display a file that has special character sequences, such as control characters or extended ASCII characters.

NOTE

The **od** command displays bytes and words of information. These display formats are not dependent upon the computer hardware.

COMMAND FORMAT Following is the general format of the **od** command.

```
od [ -bcDdFfOoSsXx ] [ file ] [ [+]offset[.][b] ]
```

BSD (Berkeley)

```
od [ -a[pP]bcdfhilovx ] [ -w[n] ] [ -s[n] ] [ file ] \
   [ [+]offset[.][bx] [ [+]label[.][bx] ] ]
```

Options The following list describes the options and their arguments that may be used to control how **od** functions.

-b Displays each byte in octal format. For example, a file (pups) containing the word PUPPIES is displayed as

```
od -b pups
000000 120 125 120 120 111 105 123 012
```

The first set of zeros is the address of the first byte displayed on the current line. The 120 is the octal representation for P.

-c Displays each byte in its ASCII character format. Nonprinting characters have the following special representations.

Notation	*Meaning*	*ASCII Label*
\0	Null	NUL
\b	Backspace	BS
\f	Formfeed	FF
\n	New-line	NL
\r	Carriage Return	CR
\t	Tab	HT
\ooo	Octal Representation of character	

All nonprintable characters that do not have a special symbol are represented using their octal representation.

-D Displays each word in unsigned decimal format.

-d Displays each long word in unsigned decimal format. For example,

```
od -d pups
000000 080085 080080 073069 083010
```

-O Displays each long word in octal format.

-o Displays each word in octal format. This is the default if you do not specify any options.

-S Displays each long word in signed decimal.

-s Displays each word as signed decimal.

-v Verbose. Displays all data. By default duplicate lines are replaced with an asterisk (*) in column 1.

-X Displays each long word in hexadecimal.

-x Displays each word as hexadecimal.

[+]*offset*[.][b] The + sign must be used if no *file* is specified. The *offset* is the number of bytes to skip before displaying information. By default the *offset* is interpreted as octal. The . (period) option changes the meaning of the *offset* to be decimal instead of octal. The b option changes the *offset* to be calculated in blocks (512 bytes) instead of bytes. Dumping continues until the end-of-file is reached.

BSD (Berkeley)

-a[pP] Displays bytes as ASCII names. If p is specified, bytes with even parity are underlined. If P is specified, bytes with odd parity are underlined.

-b Same as System V

-d Same as System V

-f Displays long words as floating point.

-h Displays words as unsigned hexadecimal.

-i Displays words as signed decimal.

-l Displays words as signed decimal.

-o Same as System V

-v Same as System V

-x Displays words as hexadecimal.

-s[*n*] Display strings of ASCII graphic characters, null terminated. The minimum string length is set to *n*, the default for *n* is 3.

-w[*n*] Display *n* input bytes on one line of output. If -w is not used the default is 16 bytes. If -w is specified without *n* the width is set to 32 bytes.

[+]*offset*[.][b][x] The same as SV with the addition of x. If x or 0x is specified, the *offset* is calculated in hexadecimal.

[+]*label*[.][b][x] Same as the *offset* option except it is intended for use with core images to indicate the real memory address.

Arguments The following describes the argument that may be passed to the **od** command.

file The name of the input file to be displayed in **od** format.

If no *file* is specified **od** reads from the standard input. Thus you can use **od** to read input from a pipe or straight off your keyboard. If you use it from your keyboard, no output is displayed until you press Return or Ctrl-D.

RELATED COMMANDS The **cat** command with the -v option performs a similar function provided by the **od -c** command.

APPLICATIONS

The **od** command can be used to view the contents of a file containing special nonprinting characters. You may find it useful to look at binary files such as compiled C programs containing machine code or data files containing data in binary format. Typically, it is used more by programmers than general users but it is one of those commands worth knowing about when you need it.

TYPICAL OPERATION

In this activity you use the **od** command to view the contents of a TERMINFO entry. Begin at the shell prompt.

1. Type **od file1** and press **Return** to display the contents of the file *file1*.
2. Now type **od -cx file1** and press **Return** to display the contents of the file in ASCII and hexadecimal format.

System V Only

3. Type **od -cx /usr/lib/terminfo/v/vt100 | pg** and press **Return**. The **od** command reformats the contents of the vt100 terminfo entry into ASCII and hexadecimal representation. Two lines of output are generated for each ten bytes read.
4. Press **Return** at the : prompt of **pg** to continue stepping through the output of **od**. Notice the definition of the *vt100* terminal is defined in this file.

---

5. Turn to Module 79 (SV), Module 81 (BSD) to continue the learning sequence.

Module 98
b pack/unpack (SV)

DESCRIPTION

The external **pack** command compresses the data stored in a file. It attempts to reduce the size of a file by compressing the data within the file. When a successful compression of data has occurred, the original file *name* is replaced by the new packed file with a name of *name*.z. Typical compression reduces the file size by 25 to 40 percent.

The external **unpack** command unpacks a previously packed file. It reverses the effects of the **pack** command. It takes a *name*.z file for input and creates an unpacked *name* file.

COMMAND FORMAT Following is the general formats of the **pack** command.

```
pack [ - ] [ -f ] [ file_list ]
unpack file_list
```

Options The following list describes the options that may be used to control how *pack* functions.

Causes *pack* to toggle the displaying of status information about the compression algorithm on or off. The first occurrence toggles the information on, the number of times a byte is used, its relative frequency, and the code for the byte are displayed. The - option may be placed between *names* to toggle the information on and off. For example,

```
pack - afile - myfile
```

would display the status information for *afile* but would not display it for *myfile*.

-f Forces the compression of the data even if *pack* realizes there is no saving of space. This may prove useful if you want all files in a directory to be packed so you don't have a mix of different file formats and names.

Arguments The following argument may be passed to the **pack** command.

file_list The name of the file you wish to pack. Multiple names may be supplied on the command line. Names must not exceed 80 characters. Two additional characters (.z) are placed at the end of each name that is packed.

The following argument is used by **unpack**.

file_list One or more files to be unpacked. The names may be *name* or *name*.z.

FURTHER DISCUSSION The **pack** command becomes more efficient the larger the files. Target files for packing are text files with more than three blocks of data. Object and binary files usually do not compress more than 10 percent, whereas text files may compress up to 50 percent.

The **pack** command may not compress the data in a file for several reasons. The following list explains each reason.

The file appears to be packed already.
The filename has more than 80 characters.
The file has links. **pack** could only change one of the filenames, leaving other filenames without the .z extension pointing to packed data.
The file is a directory.
The file cannot be opened (it's not readable).
The compression of the file will not save any disk space.
A compressed file with the same *name*.z already exists.
The *name*.z file cannot be created (you do not have write permissions).
Some type of I/O error occurred while pack was compressing the data.

NOTE

Directories cannot be packed.

The **unpack** command may fail if any of the following conditions occur.

The file does not appear to be a packed file.
The filename provided has more than 80 total characters, including the .z.
The file cannot be opened.
The unpacked version of the filename exists (name).
The unpacked file cannot be created.

RELATED COMMANDS Refer to the **pcat** command described in Module 102.

RELATED FILES The pack command displays status messages to the standard output if the -option is toggled on.

The filenames used on the command line that are to be packed must contain 80 characters or less.

The **unpack** command unpacks files with a filename of *name*.z. It can accept arguments of *name* or *name*.z.

RETURN CODES The **pack** command returns an exit status equal to the number of files it failed to pack. The **unpack** command returns an exit status equal to the number of files it failed to unpack.

APPLICATIONS

You use the **pack** command to reduce the disk space required to store your commands. Usually, you pack files that are not changed very often.

The **unpack** command is used only to unpack files previously packed by the **pack** command. Usually, you only use **unpack** if you want to make changes to the data stored in the packed file. If you only need to view the contents, you should use the **pcat** command.

TYPICAL OPERATION

In this activity you use the **pack** command to compress your copy of the password file. Begin at the shell prompt.

1. Type **cp /etc/passwd passwd** and press **Return**.
2. Type **pack - passwd** and press **Return**. The status information is displayed on your screen. Notice the **pack** command informs you on the percentage of compression gained by the pack.

In this activity you use the **unpack** command to unpack the previously packed passwd file. Begin at the shell prompt.

1. Type **unpack passwd.z** and press **Return**. The passwd file is unpacked and the filename changed to **passwd**. If you have doubts, you can **cat** the unpacked file to view the contents.
2. Turn to Module 102 to continue the learning sequence.

Module 99
PARAMETERS/VARIABLES/ARRAYS

DESCRIPTION

Variables are names that the shell uses to store values. The shell maintains four types of variables;

- Positional parameters
- Special parameters
- Named variables
- Array variables

Arrays are discussed in the Arrays section. All variables are referenced by preceding them with a dollar sign ($). Refer to the Variable Substitution and Referencing section for further information on referencing the value stored in a variable.

All variables are expanded by the shell before a command is executed. Normally, the data type of a variable defaults to *character* but the **ksh** provides a command, **typeset**, which allows you to set a variable to a certain data type. For a complete explanation of **typeset** refer to Module 145.

The Environment A very important concept to understand about the shell and most UNIX processes is their *environment*. The shell environment is a set of *exported* variables, which control how the shell functions and provides information for other programs. These exported variables are referred to as *environmental variables*. When you set a variable (assign a value to it), it may or may not be exported. If it is NOT exported, it is considered a *local* variable and is only known by the current shell. If a variable has been exported, it is considered *global* and is known by the current shell and all new child processes. Refer to Module 47 for further information on **export** command.

NOTE

Parent processes do not know about environmental variables declared in child processes. When a child process dies, all variables are lost. The concept of global descends down through child processes but not up.

Positional Parameters Positional parameters are numbered variables representing the position of arguments passed to the shell script. The name of the shell script is passed as positional parameter 0, argument 1 is passed as positional parameter 1 and so on. Positional parameters are assigned during invocation of the shell. The following example illustrates the positional parameters.

```
cj> myscript pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9
```

Inside myscript $1 would equal pp1, $2 would equal pp2, and so on up to $9.

To reference positional parameters greater than 9 you must enclose the parameter in braces. For instance, to reference positional parameter 18 you would use **${18}**.

NOTE

Positional parameters may be reset by using the **set** command. Refer to Module 118.

Special Parameters Special parameters are automatically set by the shell. The following list describes each of these parameters. To reference the value of the parameter precede it with a $.

0	The name of the current shell script
#	The number of positional parameters (arguments) passed to the shell script updated to reflect the number of parameters set by executing the *set* command.
$	The process ID of the shell script being executed
!	The process ID of the last process submitted to the background for execution
?	The exit status of the last command not executed in the background
-	The current options in effect for the shell being executed (hyphen or minus sign)
_	The last positional parameter of the last command (underscore). For example, if you type the command **cat file1 file2** you can then type a command like, **vi $_** to edit file2.
*	All positional parameters passed to the shell script being executed. Expands as a single argument, separated by the first character of the IFS variable (see Variables section). "$*" expands to "$1 $2 $3 ...".
@	All the positional parameters of the shell script being executed. Expands to separate arguments. "$@" expands to "$1" "$2" ...
ERRNO	Contains the error number of the most recent system call. The meaning of the error number is dependent upon your hardware. Look in /usr/include/sys/errno.h for a list of errors.
LINENO	Contains the current line number of a shell script being executed
OLDPWD	The previous directory before the last cd command
OPTARG	Set to the option argument following an argument when you use the **getopts** command
OPTIND	The index of the next option to be processed by the **getopts** command
PPID	The process ID of the parent process
PWD	The present working directory (current directory)
RANDOM	A random number. It is reassigned each time it is referenced. The random number sequence can be initialized by assigning an integer value.
REPLY	Used by the **select** keyword command and used when no variable is given
SECONDS	Contains the number of seconds since you invoked the current shell

---

C Shell

Remember each of the following must be preceded by a dollar sign ($) when being referenced.

0	Same as **ksh**.
?var {?var}	Is set to one if the variable is defined; otherwise 0.
$	Same as **ksh**.

<	Read one line from the standard input(stdin).
argv	Contains the list of command line arguments for the current shell. The elements of the array correspond to the positional parameters $1, $2, ...
cwd	The full path of the current working directory.
status	The status code returned by the most recent command. If the command terminated abnormally, 0200 is added to the status code. Internal commands that fail return a status of 1.

Named Variables (Keyword Parameters) The attributes of named variables are listed as follows.

- Shell variables that can have their values assigned and changed.
- Their names must begin with an upper or lowercase letter or an underscore.
- The variable name must contain only upper or lowercase letters, underscores, or digits (0-9).

The general formats for assigning or changing a value of a named variable follow.

```
named_variable=value

keyword_parameter=value command options arguments
```

The first format assigns a value to the symbolic variable name. In the second format the named parameter (keyword parameter) is assigned a value on the command line that can be referenced by variables inside the shell script. This provides for another way to pass information into the shell script. The parameter is only set for the duration of the command.

Values The value of a variable can be null or nonnull. The null character equates to nothing, no data assigned. Therefore, nonnull is any ASCII text or numeric value. Nonnull values reflect the fact that the variable is set. Null values do not reflect the fact that the variable is set, since unset variables have a value of null. An unset variable does not exist and thus reference to the variable may return an error or a null value, depending on the circumstances.

NOTE

No spaces are allowed on either side of the equal sign (=) when assigning values to variables.

Reserved Named Variables The following variables are used by the **ksh**. It is advisable to use these variables only as defined below.

CDPATH	A colon-separated list of directories used by the **cd** command. If the directory argument specified on the **cd** command line does not start with a /, then the CDPATH is searched for the argument and if a match is found, **cd** attempts to change to the directory.
COLUMNS	The width of your terminal screen in character columns. The default width is 80. Used by the **select** list and the inline edit window of the shell.
EDITOR	If the VISUAL variable is not set and the basename of this variable is set to *emacs*, *gmacs*, or *vi*, then the corresponding in-line editing mode of the shell is used. Refer to the Inline Editing Mode section of Module 50.

ENV — The pathname of a file that will be executed when a new *shell* is invoked. It is commonly used to store *aliases* and *functions*. A common filename is .kshrc.

FCEDIT — The default editor used by the **fc** command. If not set, /bin/ed is used.

FPATH — A colon-separated list of directories used by the **autoload** command. Defines a set of directories to be searched for a specified function.

HISTFILE — If set when the shell is invoked, then the pathname assigned is used to store the command history. If not set or is not write enabled, $HOME/.sh-history is used.

HISTSIZE — If set when the shell is invoked, it specifies how many previous commands may be accessed.

HOME — The default directory **cd** uses if no argument is supplied.

IFS — Internal Field Separator. The shell uses the characters in the IFS variable to split the following into fields:

- The result of command substitution.
- The result of positional parameter substitution.
- The input fields read by the *read* command.

LINES — Used by the **select** command to limit the number of lines to be used in the select list.

LOGNAME — The user name that owns the current shell.

MAIL — Identifies the file that contains your electronic mail. The shell checks this file for new mail and informs you based on the MAILCHECK variable.

MAILCHECK — Specifies how often, in seconds, the shell will check for new mail in the file referenced by $MAIL. The default is 600 seconds. If 0 is used, then the shell checks for new mail after each command is executed.

MAILPATH — Identifies a list of files to be checked for the arrival of new mail. Each file is delimited by a colon. If a % sign and text follows a filename, then the text is displayed in place of the default message "you have mail."

PATH — Identifies a list of directories to search when the shell attempts to locate a command. An empty field (::) or a field containing a period specifies the current directory.

PS1 — The primary shell prompt. Default is $. The value of PS1 is displayed each time the shell is waiting for input from the keyboard.

PS2 — The secondary shell prompt. Default is >. Displayed if the shell requires more information to complete the command syntax.

PS3 — Used by the **select** command to request a response. Default is #?.

PS4 — Displayed for execution of each command during a shell trace. The default is +. A useful setting is:

```
PS4='[$LINENO]+'
```

SHACCT — Identifies the file the shell will use to store accounting information about each command it executes.

SHELL	Specifies the shell to be used by commands when you escape to a subshell. The **vi** and **ed** command use SHELL when escaping to a subshell. It is also used during startup by the login shell to decide if it should be a restricted shell.
TERM	Identifies your terminal type. Must be set for programs that provide screen control (such as **vi**) to function properly.
TERMINFO	Identifies the directory to search for terminal information describing your terminal type.
TERMCAP	Identifies the database to scan for terminal information describing your terminal type. TERMINFO replaced termcap in System V Release 2.0. But termcap is still used by other applications.
TMOUT	Specifies the number of seconds of inactivity that will cause the shell to automatically log out of the system. If set to 0 or not set (NULL), then time-out is disabled. The message "shell time out in 60 seconds" appears on the terminal. If a key is pressed within 60 seconds the user is not logged off.
TZ	Defines the time zone displayed by the **date** command.
VISUAL	Overrides the assigned value of the EDITOR variable. Used for the same purpose.

C Shell

Variables that have a capitalized version are set by the C shell at login and reset each time the lowercase version is reset.

cdpath	Same as ksh except list of directories is separated by white space not colons.
echo	Echo each command after substitution, but before execution.
fignore	A list of filename suffixes to ignore at filename completion. Commonly set to '.o'.
filec	Enable filename completion. Thus Ctrl-D and Escape have special meaning. Ctrl-D — Print a list of all filenames that start with the preceding string. Escape — Replace preceding string with the longest unambiguous extension.
hardpaths	When set, pathnames in the directory stack are resolved to contain no symbolic-links.
histchars	A two character string. The first character replaces the default exclamation (!) as the history substitution character. The second replaces the default carat (^) as the quick substitution character.
history	The number of lines saved in the history file. The larger the number the less the memory for the csh to use. A number between 100 and 250 is reasonable.
HOME home	The user's home directory. The expansion of ~ refers to the value of this variable.
ignoreeof	If set, the shell ignores Ctrl-D(EOF) from stdin.
mail	A list of files the shell checks for mail. If the first word of the list is a number *n* then the shell checks each file every *n* minutes.

nobeep	Suppress the bell during ambiguous filename completion requests.
noglob	Inhibit filename substitution. Often used in shell scripts to prevent further filename expansion.
nonomatch	Returns the filename expansion pattern instead of an error if the pattern did not match any files for expansion. Malformed patterns still return an error.
notify	If set the shell notifies you immediately when jobs complete. By default the shell notifies you prior to displaying the next prompt.
PATH path	Same as **ksh** except the list of paths is separated by white spaces instead of colons.
prompt	Same as ksh. Often tested in you .cshrc file to know if the shell is interactive or not. if ($?prompt == 0) exit says that if the prompt is not set then exit, which assumes the shell is not an interactive shell.
savehist	The number of lines to save in the .history file when the user logs out. The more lines saved the longer it takes for the C shell to log into the system.
shell	Same as the **ksh** shell.
time	Controls the automatic time of commands. Refer to the module on the time command for a complete list of settings.
TERM term	Same as the **ksh** TERM variable.
USER user	Same as the **ksh** LOGNAME variable.
verbose	Display each command after history substitution has occurred.

Default Variable Assignment The shell assigns default values to the IFS, MAILCHECK, PATH, PS1, and PS2 variables during login. The HOME and MAIL variables are assigned default values by **login**. You may reassign values to these and any other of the reserved words, but keep in mind how the shell is going to use each variable. Most reserved word assignments are placed in the $HOME/.profile or $HOME/.kshrc. Each time you log in, your reserved variables are set to the desired values without any extra effort on your part.

ARRAYS An *array* is a simple data structure containing indexed elements that allows you to store multiple values of the same type in one *name*. Each element can be accessed by a subscript. The general formats of an array follow.

```
variable[subscript]=value          # to assign a value to an array element
${variable[subscript]}             # to reference an array element
```

Notice NO spaces are allowed in either format. *Variable* is a named variable. The brackets surround the subscript. The *subscript* may be an integer or an arithmetic expression that returns an integer. The subscript integer must be in the range of 0 to 511. The *value* must be of the same type as the array.

The ${...} must enclose the name of the array for variable referencing. If no subscript is specified, element zero is used. The following examples help clarify the use of arrays.

```
cj> num[0]='363-4804'          # assign 363-4804 to element 0 of num
cj> num[1]='282-2101'          # assign 282-2101 to element 1 of num
cj> echo "${num[0]}   ${num}"  # display num sub 0 twice
cj> echo ${num[*]}             # display all elements in num
cj> echo ${#num[*]}            # display number of elements in num
```

For more information on data types refer to the **typeset** command, Module 145. Also refer to the **let** command, Module 73, and the **export** command, Module 47, and sections on Variables and Variable Substitution and Referencing.

VARIABLE SUBSTITUTION AND REFERENCING *Variable substitution* is the referencing of the value stored in a variable name. It is performed when a variable is preceded by a $. The shell *substitutes* the value assigned to the variable in its place. Variable substitution or expansion is performed before filename expansion and before the command is separated into arguments. The following list describes the general format of variable substitution. In the following list *word* may also be a variable.

$var	Expands to the value previously assigned to the variable. The longest possible word following the $ is interpreted as the variable name.
${var}	Same as previous but controls what is interpreted as the variable name. By using the braces you can have characters follow immediately after the variable name (${var}EXT). Double digit positional parameters may be referenced using this notation(${12}).
$var[sub] ${var[sub]}	Expands to the value of the element at the subscript *sub* in array *var*. The subscript is evaluated before the variable var is expanded.
${var[*]}	Expands to all values stored in the elements of an array. Each element is separated by the first character of the IFS variable.
${#var}	Expands to the number of characters or bytes used by the value of the variable. If the variable is * or @ then the number of positional parameters are returned.
${#var[*]}	Expands to the number of elements stored in the array. Not supported by **csh**.
${var:-word}	Expands to the value of variable if value is nonnull; otherwise, expands to value of word. Not supported by **csh**.
${var-word}	Expands to the value of variable if the variable has been set (null or nonnull); otherwise, expands to the value of word. Not supported by **csh**.
${var:+word}	Expands to the value of variable if variable is nonnull; otherwise, expands to a null. Not supported by **csh**.
${var+word}	Expands to the value of variable if variable is not set (unset); otherwise, expands to value of word. Not supported by **csh**.
${var:=word}	Expands to the value of variable if variable is nonnull; otherwise, expands to the value of word and the variable is assigned the value of word. Not supported by **csh**.
${var=word}	Expands to the value of variable if variable is set; otherwise, expands to the value of word and the variable is assigned the value of word. Not supported by **csh**.

${var:?}	Expands to the value of variable if variable is nonnull; otherwise, expands and displays contents of word and exits the shell. If word is empty, a default message is used. Not supported by **csh**.
${var?word}	Expands to the value of variable if variable is set; otherwise, expands and displays contents of word and exits the shell. If word is empty, a default message is used. Not supported by **csh**.
${var#pattern}	Expands to the value of variable with pattern removed from the left side of variable. If no pattern is matched, then value of variable is used. The smallest matching pattern is removed. Pattern may contain filename generation characters (*, ?, and [...]). Not supported by **csh**.
${var##pattern)	Same as previous syntax except the longest matching pattern in removed from the variable. Not supported by **csh**.
${var%pattern}	Same as previous single # syntax except the smallest matching pattern is removed from the right side of the variable. Not supported by **csh**.
${var%%pattern}	Same as previous % syntax except the longest matching pattern is removed from the right side of the variable. Not supported by **csh**.
$*n*	Positional parameter. The arguments from the command line. $0 is the name of the command. $1 is the first argument. The last argument can be accessed by **eval \$$#**

Remember all of the Special Parameters can be used also.

C Shell

$var ${var}	Same as **ksh**. The csh allows entire array assignments to variables in one assignment. For example, **myvar=(sub1 sub2 sub3 sub4)** creates 4 elements in the array myvar.
$var[sub] ${var[sub]}	Same as **ksh**.
$var[*]	Same as **ksh**.
$var[n-m] ${var[n-m]}	Expands to the values of the elements from subscript *n* through subscript *m*
$var[$#var] ${var[$#var]}	Expands to the value of the last element of array *var*.
$#var ${#var}	Expands to the number of words stored in the variable.
$?*var* $?{*var*}	Substitutes the value 1 if *var* is set, otherwise the value of 0 is substituted.
$*n*	Same as **ksh**. The last argument can be accessed by $argv[$#var].

The **csh** can also modify the values of variables. The following list of modifiers may be added to the end of a variable to modify the value. The variable and modifier must be separated with a colon (:). For the following assume the value of VAR is /tmp/afile.ext.

:e	Remove everything but the suffix. Like using **expr $var : '.*\.\(.*\)'** but much faster and easier to remember. So ${VAR:e} returns ext.
:h	Remove the last filename or directory from the path. Like the **dirname** command. Thus ${VAR:h} returns /tmp.
:q	Quote the substituted words, preventing further substitutions. This prevents filename expansion and provides the same effect as placing double quotes around the variable. Should be used with arrays (wordlists).
:r	Remove suffix of the form ".xxx" leaving the basename. Somewhat like **basename ${VAR}** suffix, except the path is not removed. The ${VAR:r} substitution returns /tmp/afile.
:t	Remove all leading components, leaving the basename. Like the **basename ${VAR}** command. So ${VAR:t} returns afile.ext.
:x	Same as q, except words are delimited by spaces, tabs, and new-lines. It basically takes a value containing white space and converts it into separate elements of an array (a wordlist).
:gh	Globally perform :h on each element of the array variable.
:gr	Globally perform :r on each element of the array variable.
:gt	Globally perform :t on each element of the array variable.

APPLICATIONS

Variables are used to store data to be used later. They are often used to pass information to programs or shell scripts.

Arrays are used to store related information in a single variable that has multiple elements. They are best utilized when the information you need to store can be accessed in sequential increments. If you have 100 values of donated money that you need to work with multiple times in a shell script, you would store them in an array. Then accessing the values becomes easy because you don't have to remember or code for 100 different variable names. You just use one array name.

TYPICAL OPERATION

In this activity you use VARIABLES and ARRAYS to store and retrieve data, directory paths, and commands. Begin at the shell prompt.

1. Type **PS1="`uname`-${LOGNAME}>"** and press **Return**. Notice in the following display your prompt has changed to the name of your UNIX System, followed by your login name and a ">". If it did not change, type **export PS1** and press **Return**.

```
cj> PS1="`uname`-${LOGNAME}>"
host1-mylogin> _
```

C Shell

Type **set prompt="`hostname`-${USER}>"** and press **Return**.

---

2. Type **l="ls -C"** and press **Return**.

C Shell

Type **set l="ls-C"** and press **Return**.

---

3. Then type **$l** and press **Return**. Notice the output of ls -C is returned to your terminal. The shell expanded the $l into ls -C and executed it.
4. Type **arr[1]="This is one element of an array!"** and press **Return** to store a string in element one of array "arr." This only works for **ksh**.

C Shell

Type **set arr=("This is one element of an array\!")** and press **Return**.

---

5. Type **echo ${arr[1]}** to display the string you stored in step 3.
6. Turn to Module 54 to continue the learning sequence.

Module 100
passwd

DESCRIPTION

The external **passwd** command allows you to create or change the password associated with your user name. The super-user (system administrator) can change any user's password and is not prompted for the old password. As a user you can only change your password.

COMMAND FORMAT Following is the general format for the **passwd** command.

```
passwd [ user_name ]
passwd [ -df ] [ -n min ] [ -x max ] [ -w warn ] user_name
passwd [ -fl ] [ -n min ] [ -x max ] [ -w warn ] user_name
passwd -s [ -a ]
passwd -s [ user_name ]
```

BSD (Berkeley)

```
passwd [ -fs ] [ user_name ]
```

Options The following list describes the options used to control how **passwd** functions.

-d — Deletes the password for *user_name*. The *user_name* will not be prompted for a password.

-f — Force the password to expire. The *user_name* is forced to enter a new password at the next login.

-l — Lock the password entry for *user_name*. No changes may be made.

-n *min* — Set the minimum number of days between password changes.

-s — Display (show) password attributes for user *user_name*. If no *user_name* is specified, your login user_name is used. The format of the information is,

```
user-name  status  mm/dd/yy  min  max  warn
```

or, if no password aging information is present

```
user-name  status
```

where

user_name	The login ID of a user
status	The password status
PS	Passworded
LK	Locked

NP	No password
mm/dd/yy	The date the password was last changed
min	The minimum number of days between password changes. The label MINWEEKS specifies the default. It is located in the /etc/default/passwd file and is set to NULL. If *min* is greater than *max*, the user may not change the password. Always use with the -x option.
max	The maximum number of days the password is valid. The user is forced to change the password every *max* days. The label MAXWEEKS specifies the default. It is located in the /etc/default/passwd file and is set to NULL. If *max* is set to -1 then aging is turned off. If it is set to 0 then *user_name* is forced to change the password at the next login and aging is turned off.
warn	The number of days before the password expires, based on *max*, that the *user_name* will be warned.

-a	Display password attributes for all entries in the password file.
-w *warn*	Set the number of days before the password expires to notify the *user_name*.
-x *max*	Set the maximum number of days the password is valid.

BSD (Berkeley)

-f	Allows you to change the information field of the /etc/passwd file for your login. Refer to **chfn** in Module 15 for further information.
-s	Allows you to change the login shell in the /etc/passwd file for your login.

Arguments The following list describes the argument that may be passed to the **passwd** command.

user_name	A valid user name in the /etc/passwd file. The user name is the first column (colon-separated columns) of the /etc/passwd file. Only the super-user can change another user's password.

FURTHER DISCUSSION The use of passwords and their implementation is highly dependent upon your company's security policy and your system administrator. Some system administrators don't require passwords, while others require passwords, perform full accounting on every user, and implement password aging. Check with you system administrator for the requirements placed on your account.

PASSWORD AGING The **passwd** command is capable of checking the elapsed time since the password was last changed, referred to as *password aging*. If the elapsed time is sufficient, then the user is allowed to change the password. Password aging also requires that a user must change passwords after a specified amount of time. The system administrator decides if password aging is activated and how long each password is aged.

BSD (Berkeley)

Most BSD based systems do not provide password aging. You should check your reference manual for specific password implementations on your system.

PASSWORD REQUIREMENTS The following is a list of requirements that a password must meet before **passwd** will allow it as your password.

1. Must contain six characters. Only the first 8 characters are significant. The label PASSLEN specifies the maximum length for a password. It is in the file /etc/default/passwd.
2. Must contain 2 alphabetic characters (upper or lowercase letters) and at least one special character or number.
3. Must not be any circular shift combination or reversal of the user name. For example, if your user name is bill, your password could not be lbil, llbi, illb, or llib, blli, ibll, libl.
4. New passwords must have at least 3 characters that differ from the old password.

BSD (Berkeley)

Berkeley is much more lenient about passwords. The following requirements must be met for a valid password.

1. Must contain four characters. Upper and lowercase.
2. Must contain six characters if only monocase.

RELATED COMMANDS Refer to the **login** command described in Module 77 and the **su** command described in Module 127. To change your information field refer to the **chfn** command in Module 15.

RELATED FILES The passwd utility reads the /etc/passwd file to retrieve the existing password and store the new password. The /etc/shadow file is used to store secure password information. Some BSD systems support a type of shadow file.

RETURN CODE The **passwd** command returns the following return codes upon completion:

0	Successful
1	Permission denied
2	Invalid combination of options
3	Unexpected failure, the password file is left unchanged
4	Unexpected failure, the password file is missing
5	The password file is busy, try again later
6	Invalid argument to an option

The $? ($status-csh) shell variable contains the return code. So **echo $?** will display the return status.

APPLICATIONS

The passwd utility is used to create a new password for a user or change an existing password. It provides a means of user security on the system. By changing your password once or twice a month you reduce the chance of another user learning your current password. The use of passwords is dependent on your company's security policy and your system administrator's implementation of security features. It is advisable to use a password and change it every two weeks even if it's not required of you.

TIP: Your password should be some meaningless string of intermixed numbers, characters, and symbols. Names of friends, wives, husbands, etc. are easily guessed by the criminal/jerk next door, down the street, or around the world.

TYPICAL OPERATION

In this activity you use the **passwd** command to change your password. The prompts that are returned from different password programs may vary but the requested response is the same.

1. Type **passwd** and press **Return** at your shell prompt.
2. Type **imauser2** and press **Return**. Notice your password is not displayed on your screen; UNIX is reading it but does not "echo" it onto your screen. If you do not have a password, you will not see the following prompt, because UNIX knows you don't have an old password.

   ```
   cj> passwd
   Enter old passwd:
   ```

3. After **passwd** verifies that you entered the correct old password it will prompt you for a new password. Type **passwd2**, your new password, and press **Return**. Again notice passwd does not echo your new password.

   ```
   cj> passwd
   Enter old passwd:
   Enter new passwd:
   ```

4. **passwd** will require you to repeat the new password so that you know you typed it correctly. Type **wrong1** and press **Return**. This will cause **passwd** to fail.

   ```
   cj> passwd
   Enter old passwd:
   Enter new passwd:
   Re-enter new passwd:
   ```

   If the new password you typed does not match, your password will not be created or updated. Notice the following message that **passwd** returns in this situation.

   ```
   Mismatch - passwd not updated!
   ```

5. Turn to Module 7 (SV), Module 15 (BSD) to continue the learning sequence.

Module 101
paste (SV)

DESCRIPTION

The external **paste** command combines files horizontally. If you have two files, it would display the first file in the left column and the second file in the second column. The **paste** command has various functions; they include:

- Combines files horizontally (side-by-side columns).
- Changes single-column input to multicolumn output.
- Removes all new-lines from all files creating one long line (serial output).

COMMAND FORMAT Following is the general format of the **paste** command.

```
paste [ -d[list] ]  [-]file_list
paste -s [ -d[list ]  file_list
```

Options The following list describes the options and their arguments that may be used to control how paste functions.

- The standard input. Causes **paste** to read one line from the standard input.

-d*list* Delimiter. Allows you to define what character you want to use to separate columns of output. If you do not specify a *list*, the output columns are displayed immediately next to each other.

If -d is not specified, a *tab* character separates all columns.

The *list* is zero or more characters used for output delimiters, replacing the default *tab* character. The *list* must follow immediately after the -d option. If *list* consists of multiple characters, **paste** uses the first character to separate columns one and two, the second character between columns two and three, and so on. If the *list* is completely used and more columns exist, then the *list* is reused from the beginning. Thus it is a circular delimiter list.

The list may contain any of the following standard C escape sequences:

\n	A newline
\t	A tab
\\	A backslash; since a backslash is a special character you must escape it.
\0	No character at all. Same as -d without a list. But may be used in multi-character *list*. Does not actually place a character on the output.

You may have to quote the *list* to keep the shell from interpreting certain characters. For example, to pass a backslash to **paste** you would have to type -d"\\\\" or -d'\\'.

Serial output. Allows you to combine all lines of each file into one line of output. The -d option may be used to change the output delimiter. The last character of output is always a new-line.

Arguments The following describes the argument that may be passed to the **paste** command.

file_list One or more files read and displayed in the appropriate format.

DIAGNOSTICS AND BUGS The following messages may be displayed if **paste** encounters a problem.

line too long An input line is too long. The longest possible input is 511 characters.

too many files Only 12 input files can be specified; you entered 13 or more. The -s option is restricted by this limit.

RELATED COMMANDS Refer to the **cut**, **grep**, **join**, and **pr** commands described in modules 29, 60, 69, and 105, respectively.

APPLICATIONS

The **paste** command is primarily used to paste files together horizontally. That is, each file is placed in a column on the output. It may also be used to take multiple files for input and generate one long serial line of output. A serial line of output has no new-line characters. Thus it is one long line. Another use is to take a stream of input and generate multiple columns of output.

TYPICAL OPERATION

In this activity you use the **paste** command to combine two files horizontally, then as one stream of ouput. In the last activity **paste** takes input from a pipe and generates a multicolumn output. Begin at the shell prompt.

1. Type **paste first city** and press **Return**. The two files are combined line-by-line with *tabs* separating the two. Your screen should look like the following display.

```
cj> paste first city
Name     City
Someone  Austin
Anyone   Dallas
```

2. Type **paste -s -d: first city** and press **Return**. This creates a single line of output. Your screen should resemble the following display:

```
cj> paste -d: -s first second
Name:Someone:Anyone:City:Austin:Dallas
```

3. Type **ls /bin | paste - - - -** and press **Return** to list out the files in the /bin directory in 4-column output. Each - tells **paste** to read a line from standard input. Since there are four files to read from, four columns are created for output.

```
cj> ls /bin | paste - - - -
adb      ar    as    basename
.
.
.
```

4. Turn to Module 20 to continue the learning sequence.

Module 102
pcat (SV)

DESCRIPTION

The external **pcat** command displays the contents of a previously packed file. The **pack** command is used to create these files.

The **pcat** command will fail if any of the following conditions are encountered.

- The file is not a packed file.
- The filename has more than 14 characters when the .z is added.
- The file cannot be opened.

If you want to create a copy of a packed file but leave the existing copy in place, you can use shell redirection in combination with the **pcat** command as follows:

```
pcat file > newfile
```

This way you can create a new version of the unpacked file without losing the old version of the packed file.

COMMAND FORMAT Following is the general format of the **pcat** command.

```
pcat file_list
```

Arguments The following describes the argument that may be passed to the **pcat** command.

file_list One or more packed files to be displayed on the standard output. You can specify the packed files with or without the .z extension. For example, to unpack passwd.z you could use either of the two command formats.

```
pcat passwd
pcat passwd.z
```

RELATED COMMANDS Refer to the **pack** and **unpack** commands described in Module 98.

RELATED FILES The **pcat** command reads its input from packed file *name.z* and displays the text on the standard output.

RETURN CODES The **pcat** command reads an exit status equal to the number of files it failed to display.

APPLICATIONS

The **pcat** command is used to view previously packed files. It lets you view packed files without having to **unpack** the file, **cat** it out, and **pack** it again. You can also use the **pcat** command to obtain a copy of a packed file without removing the packed copy. If you use the **unpack** command, the packed version of the file is removed.

TYPICAL OPERATION

In this activity you use the **pcat** command to view your packed version of the passwd file. Begin at the shell prompt.

1. Type **pack passwd** and press **Return** to pack your copy of the *passwd* file.
2. Type **pcat passwd.z** and press **Return**. Notice the *passwd* file is displayed on your screen as it normally appears in a noncompressed state.
3. Type **unpack passwd** and press **Return** to unpack the *passwd* file.
4. Turn to Module 122 to continue the learning sequence.

Module 103
pg (SV)

DESCRIPTION

The external **pg** command is used to examine a file one screen at a time. It functions as an enhanced **cat** command or a stripped down **view** command, providing functionality somewhere in between. The **pg** paginator allows you to:

- Peruse the standard input or multiple files.
- Page forward or backward multiple pages.
- Search forward or backward for a regular expression or string.
- Scroll forward or backward by lines.
- Goto specific lines.
- Redisplay the current page.
- Move forward or backward to different files.
- Save the current file to a new file.
- Display a built-in help screen.
- Use options to change its functionality.

COMMAND FORMAT Following is the general format of the **pg** command.

```
pg [ - ] [ -cefnrs ] [ -n ] [ -p prompt ] [ +line ] [ +/pattern/ ] \
file_list
```

Options The following list describes the options and their arguments used to control how **pg** functions.

-	Forces **pg** to read from the standard input.
-c	Home the cursor and clear the screen before displaying each screen. Each page is displayed on a clear screen instead of scrolling the lines. The TERM variable must be defined. The TERMinal type must have clear screen capabilities defined in the TERMCAP or TERMINFO database.
-e	Don't prompt for a response at the end of each file. Normally **pg** prompts you at the end-of-file.
-f	Inhibits line splitting. If a line is longer than the screen width, **pg** normally splits the line. For instance, if the line has many escape sequences to control terminal attributes, **pg** will split the line and it will appear incorrect. The -f option is useful when characters that don't appear on the display are embedded in the file.
-n	Does not require a new-line character to process a command. Single-letter commands are interpreted and processed immediately.
-r	Restrict shell escape from **pg**. An error message is displayed if an attempt is made to escape to the shell.

-s Turns on standout mode causing **pg** to display its prompts and messages in standout mode. Usually, this is reverse video. The TERMCAP or TERMINFO entry for the specified TERMinal defines what standout mode is on your terminal.

-n Specifies the number of lines **pg** uses for the screen window. If the terminal is defined to have a 24-line screen, the window is set to 23 lines.

-p *prompt* Changes the prompt displayed by **pg** to be *prompt*. Normally, a colon (:) is displayed. If %d is placed in the *prompt* string, the current page number is substituted.

+line Begin displaying the file at line number *line*.

+/pattern/ Begin displaying the file at the first occurrence of *pattern*. Pattern may be a string or a regular expression.

Arguments The following describes the argument that may be passed to the **pg** command.

file_list One or more files **pg** reads and displays on your terminal screen. **pg** prompts you between each file unless the -e option is used.

FURTHER DISCUSSION After each screen is displayed **pg** prompts you for a response. There are three basic types of responses:

- Press Return to display the next screen.
- Press Del to terminate displaying the current file. If the current file is the last file then *pg* terminates. Otherwise, the next file is displayed,
- Enter a command described in the following COMMANDS section.

The **pg** command needs to know the attributes for your terminal. It searches the TERMCAP or TERMINFO database for the terminal defined by the TERM variable. If TERM is not set, **pg** assumes you are on a *dumb* terminal.

COMMANDS At the end of each screen **pg** prompts for a response. To continue you must enter one of the following commands at the prompt. The commands are separated into four categories:

- Perusal commands that provide scrolling and positioning within the current file.
- Search commands that allow you to search for a desired pattern.
- Environment commands that let you control how *pg* functions.
- Miscellaneous commands to quit, get help, save current file, or escape to a subshell.

Perusal Commands These commands let you move around the current file (line-by-line motion only) and peruse the data. An address may be placed before these commands except for the Redisplay (. or Ctrl-L) and Last Screen ($) commands.

The address affects pages or lines depending on the command. Using these commands you can move to an absolute address or a relative address. An *absolute address* is measured from the beginning of the file. A *relative address* is measured from your current position within the file.

To move to an absolute address you precede the command with an unsigned number. For example, if you want to go to line 25 of the file you type 25l. To move to a relative address you precede the command with a signed number. For instance, +25l would move you forward 25 lines. To move backward you use -25l. The default address is a relative movement of +1 for each command. In

the following descriptions *adr* refers to an address, the brackets ([]) signify that the address is optional.

[*adr*]Return	Display the page specified by address *adr*. If you use an unsigned number like 3, **pg** goes to page three. If you use +3, **pg** moves forward three pages. And -3 causes **pg** to move backward three pages. If you do not provide *adr*, +1 is assumed and the next page is displayed.
[*adr*]l	Scroll number of lines specified by address *adr*. If a number like 15 is used **pg** moves to line 15. If a +15 is is used **pg** moves forward 15 lines. And if -15 is used **pg** moves back 15 lines.
[*adr*]d [*adr*]Ctrl-D	Scroll half of the screen length. A positive number means to scroll forward. A negative number means scroll backward.

The following two perusal commands do not accept addresses:

(period) Ctrl-L	Redisplay the current screen. Use this command after using the h command or if line noise has garbled your screen (if connected via a modem).
$	Go to the last screen of the current file and display it.

Search Commands The search commands provide a means of skipping forward or backward to an occurrence of a known text pattern. The pattern may be a simple string of text or a regular expression. Regular expressions are generalized patterns that match multiple sets of text. The regular expressions used by **pg** are the same as defined in Module 39, which discusses the **ed** command.

A pattern must end with a Return, even if the -n option is used. If the pattern being searched for is not found, the next page is displayed. The pattern is designated by the *pat* string in the following commands. The following commands are used to perform searches:

[*i*]/*pat*/[tmb]	Search forward for the *i*th occurrence of the pattern *pat*. If you do not specify a number for *i*, 1 is assumed. The search starts with the next screen of data. Normally, the line containing the *pat*tern is positioned at the **t**op of the new screen. See Line Positioning in the following section for details on using **tmb**.
[*i*]^*pat*^[tmb] [*i*]?*pat*?[tmb]	Search backward for the *i*th occurrence of the pattern *pat*. Again, 1 is assumed if you do not specify *i*. The search begins on the line above the current screen. Some terminals do not handle the question mark (?) properly; if your terminal is one of these, use the caret (^) notation.

Line Positioning By specifying one of the **t**, **m**, or **b** line positioning letters you can have the line displayed in one of three positions:

t	Display the line that the *pat*tern was found on at the **t**op of the screen.
m	Display the line that the *pat*tern was found on in the **m**iddle of the screen.
b	Display the line that the *pat*tern was found on at the **b**ottom of the screen.

The last letter specified remains in effect until a new letter is used. The **t** letter remains in effect until you change it to **m** or **b**.

Environment Commands The environment commands are used to change the viewing environment of the **pg** command. You can skip forward or backward to different files or reset the window size by using the following commands:

[*i*]n — Skip forward to the *i*th next file specified on the command line. If you do not specify *i*, 1 is assumed.

[*i*]p — Skip backward to the *i*th previous file specified on the command line. If you do not specify *i*, 1 is assumed.

[*i*]w — Set the window size to *i*. The next window will be displayed. If *i* is not given, the same window size is used and the next screen of data is displayed.

Miscellaneous Commands The following commands are used to perform various functions while using **pg**.

s *filename* — Save the input in a file named *filename*. The input could be a pipe; by using the s command you can save it to disk. Only the current file (input stream) is saved.

h
?
— Displays a help screen. A summary of each **pg** command is given. To redraw the same screen of text type Ctrl-L or . (period). To display the next screen press Return.

q
Q
— Quit **pg**.

!*command* — Escape to shell. The *command* is passed to the command interpreter defined by the SHELL variable. If SHELL is not set, the default shell is used. The default is usually /bin/sh.

^\
Del
— If a QUIT or INTERRUPT signal is sent to **pg** while it is writing output, it stops sending output and displays the command prompt. The QUIT signal is generated by pressing Ctrl-\. The INTERRUPT signal is generated by pressing Ctrl-C or Del. If either of these signals are sent while **pg** is at the command prompt, **pg** terminates or begins displaying the next file if present. The display may become garbled if one of these signals is sent while text is being displayed.

DIAGNOSTICS AND BUGS If the terminal you are using does not handle tabs properly, the output may be formatted incorrectly.

If you are using **pg** as a filter with another command that changes your terminal I/O settings, the settings may not be reset properly.

RELATED COMMANDS Refer to the **cat**, **grep**, **head**, **sed**, and **tail** commands described in modules 13, 60, 63, 117, and 131, respectively.

RELATED FILES The **pg** command is a filter that reads the standard input and writes to the standard output. It can also read from a list of files.

NOTE

If the standard output is redirected to a file or pipe (it no longer is the terminal), **pg** performs much like the **cat** command. If there is more than one file, **pg** places a header at the top of each file.

APPLICATIONS

The **pg** command provides a powerful and flexible way of viewing files larger than a screen in length. It can be used in place of the **cat** or **view** command. It provides the ability to search for strings and skip lines forward and backward. It can also move between multiple files.

TYPICAL OPERATION

In this activity you use the **pg** command to find your login entry in the /etc/passwd file. Begin at the shell prompt.

1. Type **pg -cns /etc/passwd /etc/group** and press **Return**. Notice the screen is cleared and the first 23 lines of the passwd file are displayed. If your TERM variable is not defined or defined incorrectly, your screen may not be cleared.
2. Now type **/mylogin** and press **Return** at the colon prompt. The next page displayed will contain your login entry. If your passwd file is small and your login is on the first page, the search will not cause a change.
3. Type **n** and press **Return** at the colon prompt to display the first screen of the next file. The /etc/group file will be displayed.
4. Type **h** at the colon prompt. The help summary screen is displayed. To return to the same screen type a **.** (period).
5. To exit **pg** type **q**.
6. Turn to Module 90 to continue the learning sequence.

Module 104
popd (csh)

DESCRIPTION

The internal csh **popd** command removes the top directory from the directory stack, then changes directory to the new top directory. The directory stack is created using the **pushd** command. Each time you use **pushd** the given directory argument is pushed onto the directory stack. The top directory is position zero with increasing numbers for other directories in the stack.

When you **popd**, the top directory is removed from the stack and the **csh** performs a **cd** stack[0], where stack sub zero is the top directory in the stack.

The stack maintains the last entry on the stack. If you try to pop the only entry left on the stack, the **popd** command complains the stack is empty.

NOTE

The ksh implements **popd** with functions. Check with your system administrator to see if popd is available.

COMMAND FORMAT The format of the **popd** command follows.

```
popd [ +n ]
```

Options The following list describes the options and their arguments that may be used to control how **popd** functions.

+n If a plus precedes a number *n* then the *n*th entry in the stack is removed. Remember entries are numbered starting at zero.

If no option is provided, the top directory (entry zero) is removed from the directory stack and the new top entry becomes the current working directory.

TIP: Most users of the **csh** alias the **pushd**, **popd**, and **dirs** commands. For example, you might want to add the following lines to your .cshrc file.

```
alias u pushd  # use u instead of pushd
alias o popd   # use o instead of popd
alias d dirs   # use d instead of dirs
```

DIAGNOSTICS AND BUGS If the stack is empty you are notified. If you specify a number greater than the number of stack entries you are notified.

RELATED COMMANDS Refer to the **dirs** and **pushd** command described in modules 36 and 110.

APPLICATIONS

The **popd** command used in conjunction with the **pushd** and **dirs** command provides a method of changing directories quickly. You no longer have to type an entire directory path twice. Once you **pushd** to a directory, you can use **popd** to return to the directory.

TYPICAL OPERATION

In this activity you use the **popd** command to remove the top directory from the directory stack. Begin at the shell prompt.

1. Type **dirs** and press **Return** to list the current directory stack.

```
cj> dirs
~ ~ /usr/spool/uucppublic /usr/lib
```

2. Type **popd** and press **Return**. The top directory is removed from the stack and the shell changes to the new top directory.
3. Turn to Module 36 to continue the learning sequence.

Module 105
pr

DESCRIPTION

The external **pr** command formats files for printing or displaying. It reads from the standard input or from a list of files and writes to the standard output. The input is reformatted according to specified options. The formatting is page and file oriented. Textual type formatting, as done by **troff**, is not performed by **pg**. The output can be reformatted to multiple columns, with or without headers and footers, indented, and double spaced, among various other formats.

COMMAND FORMAT Following is the general format of the **pr** command.

```
pr [ [ -col ] [ -wwid ] [ -a ] ] [ +p ] [ -dfprt ] [ -eck ] [ -hhdr ] \
   [ -ick ] [ -llen ] [ -nck ] [ -ooffset ] [ -sc ] [ -F ] [ filelist ] \
   [ - ]

pr [ [ -m ]   [ -wwid ]          ] [ +p ] [ -dfprt ] [ -eck ] [ -hhdr ] \
   [ -ick ] [ -llen ] [ -nck ] [ -ooffset ] [ -sc ] [ -F ] [ file_list ] \
   [ - ]
```

The first format is used to specify the number of columns (-col) to display on the output. The second format displays the output based on the number of input files. If you have three input files, the output is displayed in three columns. The *-col* option and the -m option are mutually exclusive; that is, you can only use one or the other on the same command line.

BSD (Berkeley)

```
pr [ -col ] [ +p ] [ -f ] [ -hhdr ] [ -llen ] [ -m ] [ -sc ] [ -t ] \
   [ -wwid ] [ file_list ]
```

The *-col* option does not appear to work on some BSD type systems.

Options The following list describes the options and their arguments that may be used to control how **pr** functions.

+p Begins displaying text at page *p*. The *p* must be an integer. The page numbers are based on the -l and -h options. The default number for *p* is 1. For example,

```
pr +5 infile
```

skips the first 226 (4 pages * 56 lines per page) lines of input and displays the fifth page of input.

-col Displays the output in *col* columns. The -e and -i options are assumed. This option cannot be used in conjunction with the -m option. The input is displayed down the first column of output then down the second column. This process is repeated for each page of output. The column widths vary depending on the

page width and number of columns requested. By default, input lines that are too long to be displayed in the column are truncated. For example,

```
pr -4 infile
data        data        data        data
.
.
```

displays the data in 4 columns; each page has a header.

-a Displays multicolumn output (*-col* option) across the columns instead of down the columns. The first line of input goes in column one, the second line goes in column two, and so on. This option can only be used with the *-col* option.

-d Double space the output. Blank lines at the top of the page are dropped.

-e*ck* Expands input character *c* to *k* positions in the output. The default for *c* is a tab. The *c* must be a nondigit. The default number for *k* is 8. Therefore, the tab positions are set to 1, 9, 17, and so on. You can omit either the *c* or the *k* or both. If *k* is given, the output tab positions are set to *k* + 1, 2 * *k* + 1, and so on. For example, if you specify

```
pr -e:20 infile
```

each : (colon) found in the input line is expanded to the next tab position that is a multiple of 20. The character following the first colon is placed in column 21 if possible.

-f Use form feed character (Ctrl-L) for new pages. Normally a sequence of new-lines are used to eject a page. A pause occurs before the first page is printed. You MUST press Return to have **pr** continue.

-F Fold the input lines to fit the current line width. If used with the -a or -m options for multicolumn output lines will be folded to fit the current columns width.

-h *hdr* Use *hdr* as the text for the header instead of the filename. The -h option is ignored if the -t option is specified or the -l option is specified with a *length* of 10 or less. The -h must be followed by a space. The *hdr* may be enclosed in quotes if it contains blanks. For example,

```
pr -h "infile - Please deliver to mylogin" infile
```

places the message "infile-Please deliver to mylogin" on the top of each page.

-i*ck* Tabs replace spaces in the output where possible. The default for *c* is a tab. The *c* must be a nondigit. The default for *k* is 8; therefore, tab positions are set at 1, 9, 17, and so on. If *k* is given, then tab positions are set at *k* + 1, 2 * *k* + 1, and so on.

-l*len* Set the page length to *len* lines. The default is 66 lines. A *len* of zero (0) is reset to 66. If *len* is less than 10 then the -t option is assumed and the header and trailers are suppressed. If the headers and trailers are not suppressed, the actual lines of text displayed per page is *len* - 10. The header and trailer require 10 lines per page. Thus a normal page only displays 56 lines of text.

-m Merges and displays all input files simultaneously in multiple columns. Each input file is displayed in its own column. The most files you can specify is eight. If lines are longer than the width of the column, they are truncated. The -m option cannot be used with the *-col* or -a options.

-n*ck* Generates numbered output lines. The line numbers are formatted in *k* spaces. The default for *k* is five. If *c* is specified, then the line number is separated from the text by character *c*.

-o*offset* Offset (indent) each line by *offset* characters. The default of *offset* is zero (0).

-p Pauses before displaying the next page of output. The output must be to your terminal screen. Your terminal's bell is sounded and *pr* waits for you to press Return before displaying the next page.

-r No diagnostic warning messages are displayed if a problem occurs while *pr* is attempting to open a file.

-s*c* Separate columns of output with the character *c* instead of the appropriate number of spaces used for normal formatting. The default character for *c* is the tab. Multicolumn output is not truncated unless the -w option is specified.

-t Suppress the headers and trailers. The normal header has two blank lines: an information line followed by two more blank lines. The information line contains the Month, Day, Time, Year, Filename, and Page number. The trailer consists of five blank lines. The -t option causes the -h option to be ignored.

-w*wid* Sets the width of the output line to *wid* characters. The normal width is 72 characters. This option only affects multicolumn output (*-col* and -m options).

If no options are specified, a header and footer are added to each page of output. The header consist of five lines, with the third line containing the Month, Day, Time, Year, Filename, and Page number. The footer is five blank lines. For example,

```
pr infile
```

produces

```
Mar  16 06:57 1989 infile Page 1
First line of input.
...
```

BSD (Berkeley)

The supported options under BSD function the same as the corresponding SV options.

Arguments The following list describes the arguments that may be passed to the **pr** command.

filelist One or more files to be formatted by *pr*

file1 The first file used for the -m option. The -m option requires two input streams (files).

file2 The second file used for the -m option

FURTHER DISCUSSION You can use **pr** to format files to your terminal screen or to a printer using the **lp** (lpr-BSD) command. If you have a printer connected to the auxiliary port of your terminal, the -f option comes in handy. You can enter your **pr -f infile** command and wait for **pr** to beep your terminal. Then you can align your printer paper and activate the printer port on the terminal. Press Return to have **pr** continue displaying your file.

Reformatting the data is useful. You may have a file that has tabs in it. Because the tabs are set at every 8 spaces, the text extends off the end of your printer paper. To resolve the problem you can use *pr*. For example,

```
pr -e4 infile
```

reduces each tab to four spaces instead of eight. This way you can squeeze more tab filled text on a narrower piece of paper or screen.

You will find it helpful to use **pr** to send files to the system printer. By using **pr** you can generate formatted listings of your program source code or various other text files. For example,

```
pr -n -e4 mysrc.c | lp -dlaser
```

formats the *mysrc.c* file with line numbers and tabs reduced to four spaces. The output of **pr** is piped to the **lp** command with a destination of the *laser* printer.

DIAGNOSTICS AND BUGS All diagnostic messages are displayed after all text has been displayed.

BSD (Berkeley)

The BSD version does not display errors when the output is to your terminal screen.

RELATED COMMANDS Refer to the **cat** and **pg** commands described in modules 13 and 103.

RELATED FILES The **pr** command reads from the standard input and writes to the standard output. It can also read from a list of files. The delay bell caused by the -f and -p options is only used when the output is directed to a /dev/tty* device. Therefore, if you pipe the output of **pr** to another command, the bells and pauses are suppressed.

APPLICATIONS

You can use **pr** to format a file for printing on a printer or displaying on your terminal screen. It can also be used to reformat the data in a file. You may want the tab stops to be reset or replace spaces with tabs where possible. The most common use of **pr** is probably the default listing when you don't specify any options. This causes **pr** to generate an output listing of your file with a header at the top of each page and a page break.

TYPICAL OPERATION

In this activity you use the **pr** command to display a file using the default settings, then display the same file using columns and various other options. Begin at the shell prompt.

Now for the activity using the above files.

1. Type **pr users | uniq** and press **Return** to display the *users* file in the default format. The **uniq** command is just used to remove the multiple blank lines produced by pr.

```
cj> pr users | uniq

Mar  16 08:15 1989   users  Page 1


Barbara
David
George
Scott
Tina
```

2. Type **pr -3 users** and press **Return** to display the *users* file in three-column output.

```
cj> pr -3 users

Mar  16 08:16 1989   users  Page 1

Barbara          George          Scott
David            Jill            Tina
```

Notice the format of the columns. The data is displayed vertically.

System V Only

3. Type **pr -3 -a users** and press **Return** to display the *users* file in three-column output with the text formatted across the columns instead of up and down. The output resembles the following listing.

```
cj> pr -3 -a users

Mar  16 08:16 1989   users  Page 1

Barbara          David           George
Jill             Scott           Tina
```

---

4. Type **pr -m users mgmt** and press **Return** to display the *users* file merged with the *mgmt* file.

```
cj> pr -m users mgmt

Mar  16 08:16 1989   users  Page 1

Barbara          Andrew
David            Barbara
George           Kathy
Jill             Scott
Scott
Tina
```

5. Turn to Module 33 to continue the learning sequence.

Module 106
print (ksh)

DESCRIPTION

The internal **print** command is a sophisticated **echo** command. It is the **ksh**'s output command. The **print** command performs like **echo** unless options are used to change how it functions. Refer to the **echo** command described in Module 38.

COMMAND FORMAT Following is the general format of the **print** command.

```
print [ - ] [ -nprRsu[n] ] [ args ... ]
```

Options The following list describes the options and their arguments that may be used to control how **print** functions.

-	Ignore all following options. Causes **print** to perform exactly like the **echo** (SV) command. Same as no options.
-n	Do not write a new-line at the end of the output. Performs the same function as the \c escape sequence. Conforms to BSD **echo** -n option.
-r	Ignore the escape sequences used in the **echo** command.
-R	Same as -r. Also ignores all options except for -n.
-s	Write all arguments (the output) to the history file.
-u*n*	Write all arguments (the output) to the file with **file descriptor** *n*. No new-line is written on the output.
-p	Write all arguments (the output) to the pipe process spawned with the |& command.

The following examples will help clarify the use of these options.

```
cj> print -R -n "This \n is a prompt:"  # ignores the \n and no new-line
This is a prompt:cj>

cj> print -s "vi mmrg"                  # enter the "vi mmrg" command
                                        # into the history file

cj> print -p "Some info on the pipe"    # write to the hidden shell pipe.
```

Arguments The arguments to **echo** are whatever text you wish to write to the standard output. If no arguments are specified, then **echo** displays a blank line by writing a single new-line. You can use the following list of escape sequence characters to produce special output from **echo**:

CAUTION

You must remember that the shell interprets the \ (backslash) as a special character. Therefore, it is recommended that you place your echo arguments inside double quotes.

\b	produces a backspace on the output
\c	write the output, but do NOT terminate it with a new-line
\f	produces a form feed
\n	produces a new-line
\r	produces a carriage return
\t	produces a tab
\v	produces a vertical tab
\\	produces a backslash
\0*n*	produces the ASCII character with the octal code of *n*. Where *n* is a 1-, 2-, or 3-digit octal number representing the desired character.

DIAGNOSTICS AND BUGS When you use the \0*n* escape sequence you must always precede the *n* with a 0 (zero). This informs echo that the number is octal.

To use escape sequences you must either enclose them in double quotes, single quotes, or use two \ (backslashes).

If you need to use these escape characters, the octal equivalents can be found in Appendix G.

RELATED COMMANDS Refer to the **echo** command described in Module 38 and the **ksh** command described in Module 71.

RELATED FILES The output of **print** is sent to the standard output. Therefore, you may redirect it to a file or a pipe. It is also possible to send the output to the shell's background pipe, using the -p option. Refer to the |& (vertical bar, ampersand) definition in the SPECIAL (META) CHARACTERS section of Module 71 on the **ksh**.

APPLICATIONS

The **print** command is used to display text and variables to the user's terminal. Since the standard output is the default, output may be redirected as desired. It is often used to direct known output into a pipe. The pipe may be the shell's own background pipe or a normal pipe. It is used in shell scripts to display return codes and debugging output. You use **print** to build menus and data entry screens. The **select** command should also be studied for menu building.

TYPICAL OPERATION

In this activity you use the **echo** command to display text and variables. You also use it to output new-lines, tabs, and no new-line via escape sequences. If you don't have the ksh, skip to the next module. Begin at the shell prompt.

1. Type **print -R -n "My user name is $LOGNAME"** and press **Return**. Notice your prompt is displayed at the end of the text. The -n informs **print** not to write a new-line to the output.
2. Type **bc |&** and press **Return**. No response occurs. Make a note of the process ID displayed.
3. Type **print -p "2 * 2"** and press **Return**. Notice the string is not displayed; it is sent to the command in background (step 2).
4. Type **read -r -p ANS** and press **Return**. No response occurs.

5. Type **echo $ANS** and press **Return**. The number 4 is displayed, the result of 2 * 2 which was multiplied by the **bc** command.
6. Type **kill -9** ***pid*** and press **Return**, where *pid* is the number of the process you wrote down in step 2.
7. Type **print -s "print this"** and press **Return**. No response occurs.
8. Press **Escape**, type **k**, and press **Return**. Notice the previously entered "print this" appears on your command line. When you pressed Esc k you entered the vi in-line editor command mode. The k moved you up one line in the history file.

```
cj> Escapek ...
cj> print this              # appeared after you pressed Escapek
this                        # appeared after you pressed Return
```

9. Turn to Module 10 to continue the learning sequence.

Module 107
printenv (BSD)

DESCRIPTION

The external **printenv** command displays your current environment. Your environment consists of variables set to certain values. If you specify an argument, the value of that variable is displayed. If the variable does not exist, **printenv** does not display anything and returns an exit status of 1. If the variable exists, the exit status is 0.

COMMAND FORMAT The format of the **printenv** command follows.

```
printenv [ variable ]
```

Arguments Only one argument may be passed to the **printenv** command.

variable The name of a variable which exists in the environment. If no variable name is provided, then all variables defined are displayed.

RELATED COMMANDS Refer to the **ksh** command described in Module 71.

APPLICATIONS

The **printenv** command is useful for displaying your current environment. You can display all variables in your current shell environment or display a specific variable. You will need to verify what a certain variable is set to while working on UNIX. One common variable you need to verify if **vi** fails to work properly is the TERM variable.

TYPICAL OPERATION

In this activity use the **printenv** command to display your environment. Begin at the shell prompt.

1. Type **printenv** and press **Return** to list out your current environment.

```
cj> printenv
ENV=/u1/ts/mylogin/.kshrc
FCEDIT=vi
HOME=/u1/ts/mylogin
LOGNAME=mylogin
MAIL=/usr/mail/mylogin
PATH=:/u1/ts/mylogin/bin:/bin:/usr/bin:/usr/lbin:/usr/local/bin
SHELL=/bin/ksh
TERM=vt100
TMPDIR=/tmp
TZ=CST6CDT
VISUAL=vi
```

2. Turn to Module 145 to continue the learning sequence.

Module 108
Programming Constructs — Shell Programming

DESCRIPTION

Shell programming constructs are used to build shell scripts. In this module we discuss the use of various shell programming constructs and the basics of creating executable shell scripts.

Before you name your shell script, verify that is does not already exist on the system. You can use the **whence**, **type**, or **which** command, depending on your login shell, to see if your current PATH can locate a command with the same name. If no command is found, then you can use the name you selected.

If you work in a heterogeneous environment, you will probably want to locate the shell with the most features common to all systems and program to that standard. A *heterogeneous environment* consists of several types of hardware and various versions of UNIX. A *homogeneous environment* consists of at least all the same versions of the operating systems. The hardware may or may not be different. So you can have heterogeneous or homogeneous hardware mixed with heterogeneous or homogeneous software.

If you write shell scripts that are used on all of the systems, you will want to consider the problem of porting and maintaining the scripts. Although you may not be able to use the latest and greatest shell features of one system, by using a shell common to all systems, you will make life easier for yourself and the next person who has to perform your job.

NOTE

We only cover the constructs of the **ksh** in this module. The **sh** is a subset of the Korn shell. The **csh** shell is not commonly used for shell programming, thus it is not discussed. I am sure csh lovers will disagree! The constructs are listed next to the ksh constructs for those who believe they must use the csh to program with.

NOTE

If the first character of a file is a # (number sign) the script is executed as a **csh** script. If it is not a # the script is executed using the **/bin/sh** shell. You can also specify a type of interpretor by using *#!interpretor* on the first line of the script file. Where *interpretor* might be /bin/sh, /bin/ksh, /bin/csh, /bin/awk, etc.

COMMAND FORMAT Following is the general format of a shell script command.

```
shell_script_name [ options ] [ arguments ]
```

Since you create a shell script you select the name. The options and arguments depend on your implementation of the programming solution.

Options Since you create a shell script you decide what options are needed. When processing options within a shell script you should always use the **getopts** command when it is available. The **getopts** command is supported on System V Release 3 and later revisions.

BSD (Berkeley)

Most BSD systems DO NOT support the **getopts** command. Therefore, you have to process the options using a **while** or **for** loop.

Arguments Arguments should also be processed using the **getopts** command. Again you write the script, so you decide what arguments are valid.

A LOCAL BIN DIRECTORY Before you start creating local shell scripts you might consider making a directory to store your shell scripts. For example,

```
cd
mkdir bin
chmod 755 bin
```

creates a directory named bin in your HOME directory. You will need to update your .profile to have the new directory in your PATH. For example, edit your $HOME/.profile, search for the last occurence of PATH (:g/PATH/p), and add the following line.

```
PATH=$PATH:~/bin              # For Korn shell
PATH=$PATH:$HOME/bin          # For Bourne shell
```

This adds your personal bin directory to your path. Now **cd** to bin to create new shell script commands.

CREATING A SHELL SCRIPT The following steps outline the creation of a simple shell script.

NOTE

Don't forget that the shell offers the ability to alias commands. Since aliases are much faster than shell scripts you might consider using aliases instead of one- or two-line shell scripts.

To create a shell script you simply enter the editor of your choice. The filename should be the name you want to use to invoke your shell script. For example, let's say you want a shell script to list all users in your group. You might want to call the command "lg" for list groups. To create the script invoke **vi** with lg as an argument.

```
vi lg
```

Once in the editor, you write the necessary commands to perform the desired task. In this case the commands would look something like the following code.

```
OIFS=$IFS
IFS=":"
set `grep $LOGNAME /etc/passwd`
GID=$4
awk -F: '$4 == "'${GID}'" { printf "%15s %5s\n", $1, $4 }' /etc/passwd
IFS=$OIFS
```

The OIFS variable is set to the current value of IFS (space, tab, new-line). Then IFS is reset to a colon (:). A colon is the field separator in the /etc/passwd file. Then you grep the user name from the /etc/passwd file. Perform a set command so the fields in the line you greped are set to positional parameters ($1, $2, ...). Then set the GID variable to $4, which is the group ID field. Now you run awk on the /etc/passwd file searching for lines where field four ($4) matches the saved group ID (GID). If a match occurs, the user name ($1) and group ID ($4) are printed.

After you have edited the text, exit the editor. Change the modes on the file to make it executable. For example,

```
chmod 755 lg
```

or

```
chmod +x lg
```

sets the mode to rwxr-xr-x. Now you and all other users can execute the command **lg**. But only you have write permission.

SHELL PROGRAMMING CONSTRUCTS

The Korn shell provides various commands to allow for structured programming. These include conditional, iteration, and grouping commands, plus the use of functions to allow for high level language type programming.

ARITHMETIC EXPRESSIONS The Korn shell has a built-in command to handle arithmetic expressions. The command is **let** which is discussed in Module 73. The Bourne shell does not have a built-in arithmetic command; therefore, the **expr** command is used. It is discussed in Module 48.

CONDITIONAL COMMANDS (Program Flow Control) Conditional commands allow you to execute a list of commands based on the result of testing an expression.

if elif else

The **if** statement is used to check the return code of a command list. If the return code is zero (0) the commands inside the **if** statement are executed. If the return code is nonzero the **elif** or **else** part of the statement is performed.

COMMAND FORMAT The format of the **if** statement follows.

```
if command_list_1
then
        command_list-2

[ elif command_list_3
  then
        command_list_4 ]

[ else
        command_list_5 ]
fi
```

C Shell

```
if (expr) then
     command-list
else if (expr2) then
     command-list
else
     command-list
endif
```

Notice that the **elif** and **else** statements are not required in the if construct.

Alternative Flow Control The shell offers an alternative way to handle simple **if then** type statements. Instead of using if statements you use the conditional symbols || and &&. For example,

```
if [ -r "$FILE" ]
then pg $FILE
fi
```

could be written as

```
[ -r "$FILE" ] && pg $FILE
```

The && symbol executes the second command if the return code of the first command is zero. Conversely, the || symbol executes the second command only if the return code of the first command was nonzero. For example,

```
[ -d "$FILE" ] || pg $FILE
```

executes the pg command if the value of $FILE is not a directory. So you could write

```
[ -r "$FILE" ] && [ -d "$FILE" ] || pg $FILE
```

which executes the pg command if the value of FILE is readable and is not a directory. Another way to write this is:

```
[ -r "$FILE" ] && [ -f "$FILE" ] && pg $FILE
```

This executes the pg command if the value of FILE is readable and is a file.

case

The **case** statement is used to perform multiple choice branches. The case statement is better suited to handle the comparison of strings than the if statement. But it is not suited for using the return codes of commands, such as the **test** command.

COMMAND FORMAT The format of the **case** statement follows.

```
case word in
 pattern) command_list ;;
 ...
esac
```

C Shell

```
switch (word)
  case pattern:
    command-list
  default: breaksw
endsw
```

The **case** statement takes the value of *word* and compares it to the list of *patterns*. The *word* may be a variable, output from a command substitution, or a constant. The *command_list* following the first pattern that matches word is executed. If no match is made, then the case statement is passed over as if it does not exist. It is advisable to enclose *word* in double quotes. This prevents the shell from aborting if *word* is nonexistent or NULL.

The **case** is well-suited for processing options within a shell script. For example, the following code can be used to process the **-ffile** or **-f file** option and option argument.

```
while [ -n "$*" ]
 do
  case "$1" in
    -f) shift; FILE=$1 ;;
   -f*) FILE=`expr $1 : '..\(.*\)' ;;
  esac
  shift
 done
```

In the **case** statement if the -f option is separate from the filename, it is **shifted** out of the way and the filename is assigned to the FILE variable. If the filename is next to the -f option, the case matches it with the -f* and executes the **expr** command to strip off the -f part before assigning the filename to FILE.

In the Korn shell you can replace the **expr** command with the **let** command and with variable substitution commands. To perform the previous **expr** command you would use

```
FILE=${1#-f}
```

The #-f removes the -f from the beginning of the $1 variable. This is much faster and easier to remember than the old **expr** command.

ITERATION COMMANDS (Loops) Iteration commands allow you to execute a list of commands while an expression remains true. When the expression returns a false value the loop is terminated.

for

The **for** loop executes a specific number of times based on the number of words in its word list. If no word list is given, the positional parameters are processed. It provides a simple way to step through lists of information.

COMMAND FORMAT The format of the **for** loop follows.

```
for NAME in wordlist
 do
  commands
 done
```

C Shell

```
foreach NAME (wordlist)
  commands
end
```

The **for** loop provides a simple means of processing lists of information. It executes a set of commands for each word in the list. Once the list of words is exhausted the loop automatically ends.

For example, the following code illustrates a **for** loop processing the output of an **ls** command. This shell script interactively moves files from a specified directory to another directory. You must specify the directory you want to move files from and the destination directory. The source directory is $1 and the destination directory is $2.

```
for FILE in `ls $1`
  do
        echo "Move $FILE to $2: \c"
        read ANS
        case "${ANS}" in
          [qQ]) echo "Leaving the for loop!"
                break ;;
          [yY]) if mv ${FILE} $2
                then echo "$FILE has been moved!"
                else echo "The move FAILED for $FILE"
                fi ;;
             *) echo "$FILE not moved!" ;;
        esac
  done
        echo after the for loop perform the next shell command
```

The output of the **ls** command is passed to the **for** loop. The **for** loop assigns the first value in the list from the **ls** to the variable named FILE. Then the appropriate statements are executed within the do and done statements until all output from the **ls** command is processed. You are prompted for a response by the first echo statement. If you answer with a **q** or **Q**, the **for** loop is exited via the break command. If you answer with a **y** or **Y**, the file is moved and the next file is processed. If you answer with any other response, the file is not moved and the next file is processed. Once all files have been processed the **for** loop automatically exits.

while

The **while** loop provides more control over command processing. It can be used to perform simple number control loops, process commands if another command is true, or process lines of information passed to it via a pipe. As long as the first command list returns a return exit code the **while** loop will continue to execute.

COMMAND FORMAT The format of the **while** loop follows.

```
while command_list_1
 do
  command_list_2
 done
```

C Shell

```
while (expr)
  command-list
end
```

The **while** loop is used for multiple types of flow control. It is a highly flexible construct since it terminates based on the exit status of the last simple command in the first command list. This allows you to execute other commands based on the completion of commands repeatedly.

For example, the following **while** loop is very simple. It displays the numbers between 1 and 100. It is written as though it were a shell script. We assumed the name of **cnt**.

```
cj> cat cnt
CNT=${1:-1}
HI=${2:-100}
while (( CNT <= HI ))
 do
   echo $CNT
   (( CNT = CNT + 1 ))
 done
```

The first line sets the variable CNT to the first parameter. If the first parameter is not provided then CNT is set to 1. The second line sets HI to the second parameter. If the second parameter is not given then HI is set to 100. So by default this loop counts from 1 to 100.

The ((CNT <= HI)) is the *Korn shell's* arithmetic command (a form of the **let** command). It checks for the condition of CNT being less than or equal to HI. The condition is true until CNT = 101. The **echo** statement displays the value of CNT. The next arithmetic statement increments the value of CNT by 1. The result of this loop is a list of numbers on your screen from one to a hundred.

For comparison in speed, you can replace the ((CNT <= HI)) statement with **expr $CNT \<= $HI > /dev/null**. Replace ((CNT = CNT + 1)) with **CNT=‘expr $CNT + 1’**.

Another example of the **while** loop shows how you can process a stream of data from a pipe command.

```
OIFS=$IFS
IFS=:
echo "USERNAME\tUSRID\tGRPID\tHOMEDIR"
cat /etc/passwd | \
while read USERNAME PASSWD USRID GRPID DESC HOMEDIR LOGINSHELL
 do
   echo "$USERNAME\t$USRID\t$GRPID\t$HOMEDIR"
 done
IFS=$OIFS
```

This script produces a simple listing of all login entries in the /etc/passwd file. The first line saves the IFS (internal field separator) to the variable OIFS. The second line sets IFS to a colon. The colon is used in the /etc/passwd file to separate fields of information. The third line displays a header for the output. The **cat** command pipes the contents of the /etc/passwd file into the while loop. The **read** statement reads one line of input at a time. Each field of the passwd file is set to its respective variable name on the **read** line. Then the **echo** statement displays the values of the current line that has been read.

until

The **until** loop reverses the termination condition used in the while loop. Instead of exiting when a return status of nonzero is returned from *command_list_1*, the until loop exits when a return status of zero is returned.

COMMAND FORMAT The format of the **until** statement follows.

```
until command_list_1
 do
  command_list_2
 done
```

C Shell

```
label:
  commands
goto label
```

The **until** loop provides a way to loop while a condition is false or until the condition is true. This is useful when you need to ensure that some other process you have no control over has finished.

```
FILE=${1:-proc1.out}
until [ -f $FILE ]
 do
  sleep 600
 done
```

In this script we set FILE to be the first parameter from the command line or to the name "proc1.out." The until loop checks for the specified FILE. If the file does not exist, the internal part of the loop is executed. Thus if the file does not exist, the shell script sleeps for 600 seconds (10 minutes). After ten minutes a check is made to see if the file exists. If the file does exist the loop terminates.

You should not use the **until** loop in situations were you are waiting on a child process in the background to finish. In this situation use the **wait** command.

break

The internal **break** command is used to break out of a **for**, **until**, or **while** loop. You can use the **break** command to exit from within a loop but remain in the shell script.

COMMAND FORMAT Following is the format of the **break** statement.

```
break [ n ]
```

C Shell

```
break [break . . .]
```

Arguments The following describes the argument that may be passed to the **break** command.

n Break *n* levels of loops. If you are nested inside more than one loop, you can use the break command to exit out to a higher level.

When the shell executes the **break** command, control immediately exits the loop. The shell continues execution as normal on the line after the loop's **done** delimiter.

The **break** command is normally used to exit infinite loops. Usually, some condition occurs, such as an error, or all processing has completed, which causes the **break** to exit the loop.

For example, the following code illustrates an infinite loop using a **break** statement to exit.

```
while true
 do
  echo "Enter command: \c"
  read CMD
  case "${CMD}" in
   break) echo Leaving the while loop!
          break ;;
         *) eval ${CMD} ;;
  esac
 done
echo next shell command
```

If you type break and press Return, the **break** statement breaks out of the loop.

continue

The internal **continue** command is used in shell scripts to skip the remaining lines in a **for**, **while**, or **until** loop. The commands that would normally be executed in the proper sequence are skipped and the next iteration of the loop is executed from the top.

COMMAND FORMAT Following is the general format of the **continue** command.

```
continue [ n ]
```

C Shell

```
continue
```

Arguments The following argument may be passed to the **continue** command.

n The next iteration of the *n*th loop enclosing this loop. For example, if you specified **continue 2**, then execution would return to the next iteration of the loop that encloses the loop you are currently in.

When the shell executes the **continue** command, control immediately skips to the next iteration of the loop. If *n* is specified, then control is passed to the next iteration of the *n*th loop surrounding the current loop.

Usually, some condition occurs, such as an error or the process is not possible, that causes the **continue** to skip to the next step in the loop. For example, the following code illustrates a loop using a **continue** statement to skip processing if a file is not readable.

```
for FILE
 do
  if [ ! -r "${FILE}" ]
  then echo "${FILE} is not readable!"
       continue
  fi
  #
```

```
        # continue further processing
        #
        cp ${FILE} ${FILE}.old
      done
```

The **for** loop processes all positional parameters. The **if** statement checks for read permission on the current file. If the file is not readable, the next file is processed. The **continue** statement forces the next iteration of the **for** loop. If the file is readable, the remaining commands in the for loop are executed.

function

Shell functions provide the ability to write structured high level programs. Using functions can reduce redundant code and provide a more understandable and elegant shell script. Before functions were supported by the shell you had to pass parameters to subshell scripts. This worked but there were limitations and it was much slower.

You want to implement functions for code used repeatedly. For example, you might want to perform screen control. So you should create a **cls** and **cursor** function at the beginning of your program. If you use a set of functions in more than one shell script, you might consider placing the functions in a file by themselves. Then, at the beginning of each script that needs to use the functions, perform the . (dot) command on the file.

COMMAND FORMAT Following is the format of a function.

```
function name { command_list ;}
functions
```

The command list can be multiple lines of shell commands, including calls to other functions. If the } is on a line by itself you do not use the ;.

By using functions you can write complex shell scripts that perform well. You reduce the need to call subshells and keep most of the redundant activities in the current shell.

A few examples of functions you might find helpful in your scripts follow.

```
function cls { # clears your terminal screen
               CLS=${CLS:-`tput clear`}
               print - "${CLS}\c" ;}

function cursor { # positions the cursor on your terminal screen
   case "$TERM" in
      ansi|vt*) echo "\033[${1};${2}H\c" ;;
     adm*|tvi*) typeset -8 ROW=COL=0
                (( rrow = $1 + 8#037 ))
                (( ccol = $1 + 8#037 ))
                r="0"${crow#"8#"}
                c="0"${ccol#"8#"}
                eval print - "\\\\033=\\\\${r}\\\\${c}\\\\c" ;} ;;
         wyse*) function cursor { echo "\033a${1}R${2}C\c" ;} ;;
   esac ;}

# set your shell prompt to present working directory
alias cd=_cd
function _cd { cd ; PS1="cj:${PWD}-> " ;}
return
```

The internal **return** command is used to exit from a shell function without exiting the current shell. The concept is the same as the return function used in the C programming language. The **return** causes execution control to return to the level that called the function. A return code is always returned to the calling level.

COMMAND FORMAT Following is the general format of the **return** command.

```
return [ n ]
```

Arguments The following argument may be passed to the **return** command.

n Used to set the return code of the function to *n*.

If *n* is not specified the return code of the last command is returned.

The **return** command causes termination of a **function** without terminating the current shell. It is useful in situations where you want to exit from a function based on a certain condition.

The **return** command can return specific return codes. You may find it beneficial to use the return codes in conditional statements. For instance, you may want to decide how to control a loop or **if** statement based on the return code of a function. The following code shows an example of a function with the possibility of returning different return codes:

```
function filechck {
   if [ $# = 0 ]
     then return 0        # if no file passed to filechck
   elif [ -r "$1" ]
     then return 4        # if file is readable
   elif [ -w "$1" ]
     then return 2        # if file is writable
   elif [ -x "$1" ]
     then return 1        # if file is executable
   fi
}
# return execution control to previous calling level
```

Each **return** statement causes the function to exit and return execution control to the shell or function that called it initially. All remaining code in the function is skipped.

NOTE

If the **return** command is used at the first level of execution (not inside a function), the shell will exit as if you used an **exit** statement.

NOTE

If you have set the exit **trap** to 0 (**trap 'command' 0**), it is executed before the function is exited. The Bourne shell function does not execute the trap command when exiting a function.

DIAGNOSTICS AND BUGS The most common problem with shell scripts that fail to execute properly is syntax problems. And most syntax problems relate to missing quotes. You will receive strange error messages if you forget to match quotes correctly.

RELATED COMMANDS Refer to the **ksh** command described in Module 71.

RETURN CODES You control what exit status is returned by using the **exit** command or the **return** command inside functions. If you do not specify a return code, the code from the last command is used.

APPLICATIONS

A shell script is commonly used to perform tasks that require the combination of multiple UNIX commands. For example, disk backup scripts are often used to perform daily backups. Instead of writing an entire program in a programming language like C, you would use a shell script. The script is much easier to program and update.

TYPICAL OPERATION

In this activity you create a shell named **phone**. Use the **break** command to break out of a **while** loop if a certain time is reached. Make the shell accept an argument for the name of someone. Begin at the shell prompt.

1. Check for an existing shell with the same name. Use the **whence** (ksh), **type** (sh), or **which** (BSD) commands. Type **whence phone** and press **Return**. If the **phone** command is not located continue to the next step. If a command with this name was located, you will need to select a new name for the shell before continuing.
2. Type **cd ~/bin** to change directories to your bin directory.
3. Type **vi phone** and press **Return**.
4. Type **i** to enter insert mode.
5. Type the following lines and press **Return** after each line.

```
cd                      # cd to your home directory
PERSON=$1               # set person to argument one.
OIFS=$IFS               # SAVE the Input Field Separator
IFS=":"                 # reset IFS to a:
cat db/phone | \        # pipe the phone file thru a while loop
while read LINE         # read one line at a time
 do
  set $LINE             # set the positional parameters to the input line
  if [ "$1" = "$PERSON" ]
  then
        NAME=${NAME:=$1}
        NUMB=${NUMB:=$7}
        echo "Name    : $NAME"
        echo "Phone #: $NUMB"
  fi
 done
 IFS=$OIFS              # set IFS back to original value
```

C Shell

There is no easy way to do this exact process in the csh. In fact a better way to do this same function is to use nawk or use the following script.

```
grep "$1" db/phone | cut -d:-f1,7 | sed 's/:/<tab/g'
```

This will work for any shell. Although on BSD systems that do not support cut you will have to replace it with an nawk equivalent.

---

6. Save the shell to disk by pressing **Escape** and typing **ZZ**.
7. Change the permissions so the shell can be executed. Type **chmod 755 phone** and press **Return**.
8. Now you are ready to execute the shell. Type **phone anyone** and press **Return**. Notice the Phone # is empty. You will need to update your db/phone database to contain valid names and phone numbers. Currently, your database does not have a "phone" field. Simply add a colon at the end of each line and place a phone number in the new field.
9. Turn to Module 73 to continue the learning sequence.

Module 109
ps

DESCRIPTION

The external **ps** command is used to report process status. A process is a single task the computer is executing. When you execute a program such as **cat**, your shell spawns a new process (task). On UNIX there are multiple tasks waiting for execution. Each task is assigned a process ID (PID). Using the **ps** command you can display information about these various processes.

Without options the **ps** command reports the process ID, the associated tty, the cumulative execution time, and the command being executed. Various options to **ps** cause more detailed information to be retrieved and select certain processes to be reported.

COMMAND FORMAT Following is the general format of the **ps** command.

```
ps [ -acdefjl ] [ -g grplist ] [ -n name ] [ -p proclist ] \
   [ -s sesslist ] [ -t ttylist ] [ -u uidlist ]
```

BSD (Berkeley)

```
ps [ -acegklnsuvwxUn ] [ -ttty ] [ name ] [ core ]
ps [ -acegklnsuvwxUttty n ] [ name ] [ core ]
```

The *n* option must be last.

Options The following list describes the options and their arguments that may be used to control how **ps** functions.

-a Displays information about all processes except process group leaders and processes not associated with a terminal.

-c Display information about scheduler priorities.

-d Displays information about all processes except group leaders.

-e Displays information about all processes currently in the system's process table. All active processes.

-f Displays a "full" listing of header information. The columns listed are:

UID PID PPID C STIME TTY TIME COMMAND

See the Column Header section that follows for complete definitions of each column header. This option only changes the number of columns displayed. It does not determine which processes to list.

-j Display session IDs and process group IDs.

-l Displays a "long" listing of header information. The columns listed are:

F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME COMMAND

See the Column Header section that follows for complete definitions of each column header. This command only changes the number of columns displayed. It does not determine which processes to list.

-g *grplist* Display only the processes whose process group leader's IDs are provided in the *grplist*. The list can be comma separated with no spaces or enclosed in quotes and space separated.

-n *name* Use the file *name* as an alternate system executable file in place of the default /unix executable. This is the bootable UNIX kernel file. This file contains the names and addresses of variables used in the system kernel. The **ps** command needs the information held in these variables. Using a different name allows you to obtain information from a previous version of the operating system that has been saved in a system crash file.

-p *proclist* Display only the processes whose process IDs are given in the *proclist*. The list can be comma separated with no spaces or enclosed in quotes and space separated.

-s *sesslist* Display information on all session leaders listed.

-t *ttylist* Displays only the processes associated with the terminals given in *ttylist*. Terminal names are the device's filename; for example, "tty05" or "co." If the name begins with "tty" then only the number is required. For example, "14" is interpreted as "tty14." The list can be comma separated with no spaces or enclosed in quotes and space separated.

-u *uidlist* Displays only the processes whose user ID (UID) or login name is given in *uidlist*. The UID is displayed in the **ps** listing unless you use the -f option; then the login name is displayed.

If no options are provided **ps** defaults to a minimal listing, displaying only processes associated with your terminal. The header columns displayed are:

PID TTY TIME COMMAND

BSD (Berkeley) options

The fields displayed by some **ps** options vary among BSD based vendors. The following provides information for the actual BSD implementation.

-a Display information about all other user's processes associated with terminals.

-c Display the internally stored command name, instead of the the command name from the process's address space. This is more reliable since the process can destroy the command arguments in its own address space.

-e Display the process's environment in addition to the arguments. The fields displayed are,

PID TT STAT TIME COMMAND

See the COLUMN HEADER section that follows for complete definitions of each column header. This command only changes the number of columns displayed. It does not determine which processes to list.

-g Display all processes. Normally **ps** only displays "interesting" processes. "Uninteresting" processes are process group leaders. Thus top level command interpreters (shells) and terminal monitoring processes (gettys) are eliminated.

-k Changes the default core file used by **ps**. The name is /vmcore instead of /dev/kmem and /dev/mem. The name must be the last arguement, *core*, on the command line.

-l Display a long listing. The following fields are displayed:

F UID PID PPID CP PRI NI ADDR SIZE RSS WCHAN STAT \
TT TIME COMMAND

See the Column Header section that follows for complete definitions of each column header. This command only changes the number of columns displayed. It does not determine which processes to list.

-n Converts certain fields to numeric output. The WCHAN output is converted to numbers in long listings. The USER field is converted to UID number fields in user listings.

-s Display the size of the kernel stack for each process. The SSIZ field is added to the output.

-t*tty* Displays processes whose controlling tty is *tty*. If the the tty is 03 then use -t03, if it is the console use -tco.

-u Display user related information. The fields are,

USER PID %CPU %MEM SZ RSS TT STAT TIME COMMAND

See the Column Header section that follows for complete definitions of each column header.

-v Display virtual memory attributes. The fields are,

PID TT STAT TIME SL RE PAGEIN SIZE RSS LIM TSIZ \
TRS %CPU %MEM COMMAND

See the Column Header section that follows for complete definitions of each column header.

-w Display output in 132 columns instead of 80. If -ww is given as an option, then an arbitrary output length is used to accommodate the length of the actual output.

-x Display all processes, even those not attached to terminals.

-U Update the private database used to store **ps** information. The **ps -U** command is often included in the /etc/rc file.

n Display only the process whose process is *n*. This option must be the last option on the command line.

If no options are provided **ps** defaults to a minimal listing, displaying only processes associated with your terminal. The header columns displayed are:

PID TT TIME STAT COMMAND

ARGUMENTS The **ps** command has two possible arguments.

name The file containing the kernel's namelist.

core The name of the file containing the in-core memory image of the process. Usually /dev/mem or a file named "core."

COLUMN HEADERS The following table defines each of the column headings displayed by **ps**. The number of columns displayed are controlled by the -f and -l options.

System V Headers

Column Header	*Description*
ADDR	The memory address of the process.
C	The processor utilization used for scheduling purposes.
COMMAND	The command name used to start the process. If the -f option is used the entire command line is given.
F	Flags associated with the process. The number on the *ps* listing is a sum, in octal, of the following flags.

Flags	*Description*
01	The process is in core memory.
02	The process is a system process.
04	The process is locked in core.
10	The process is being swapped.
20	The process is being traced by another process.

Each system may implement a different set of flags; therefore, you should check you system's User Reference Manual for specific flags for your UNIX System.

PRI	The priority of the process. The lower the number the higher the priority.
NI	The "nice" value used to calculate the priority. Can be set using the **nice** command.
PID	The process ID of the process. This is the number you use to kill a process with the **kill** command.
PPID	The process ID of the parent process.
S	Displays the status of the process. The following table defines the possible status values.

Status	*Description*
I	Idle or Intermediate: The process is being created.
O	Operating: the process is running on a processor.
R	Running: the process is in a run queue.
S	Sleeping: the process is waiting for an event to complete, such as I/O, etc.
T	Traced: the process is stopped by a signal because the parent is tracing it.
W	Waiting: the process is waiting for a resource to become available; it is swapped out.
Z	Stopped: the process has terminated and is waiting for its parent to allow termination ("zombie").

STIME	The time the command started. Given in hours, minutes, and seconds. A process started over twenty-four hours ago is given in months and days.
SZ	The size of the process. It is the amount of memory used by the process in (512 or 1024 byte) blocks.
TTY	The terminal from which the process was started. A ? is displayed if there is no controlling terminal.
TIME	The cumulative execution time for the process.
USER	The name of the user who owns the process.
UID	The user ID number of the process's owner. The login name is displayed when you use the -f option.
WCHAN	The address of an event for which the process is waiting. If the process is executing, the field is empty. Refer to the /usr/include/sys/proc.h file for descriptions of the addresses.

BSD (Berkeley) Headers

Column Header	*Description*
%CPU	Percentage of CPU utilization of the process. The result of dividing the decaying average by the previous (real) time. The total %CPU of all processes often exceeds 100% because very young processes have very small time bases used for division.
%MEM	Percentage of real memory used by this process.
ADDR	The swap address of the process.
CP	Same as SV C.
COMMAND	Same as SV COMMAND.
F	Flags associated with the process. The number on the *ps* listing is a sum of the following flags.

Flags	*Description*
000001	The process is in core.
000002	The process is a swapper or pager.
000004	The process is being swapped out.
000008	Save the area flag.
000010	The process is being traced.
000020	The parent has been told that this process has stopped.
000040	The user has set a lock in core.
000080	The process is in page wait state.
000100	A flag to prevent swap out.
000200	Delay the unlock of pages.
000400	The process is working on exiting.
000800	The process is performing physical I/O.
001000	The process is a result of a vfork.
002000	Another vfork flag.
004000	No vm, the parent process is in a vfork.
008000	The data space is initialized on demand from inode.
010000	The system detected anomalous vm behavior.
020000	The user has been warned of anomalous vm behavior.
040000	The process is timing out during sleep state.
080000	The detached process was inherited by **init**.
100000	The process is using old signal mechanism.

LIM	A soft limit imposed on the memory used. If no limit has been set, xx is displayed.
NI NICE	Same as SV NI.
PAGEIN	The number of disk I/Os caused by references to pages of the process not loaded in main (core) memory.
PPID	Same as SV PPID.
PRI	Same as SV PRI.
RE	Residency time of the process. The amount of time spent in core in seconds.
RSS	The size of the resident set of the process (in 1024 byte blocks). The real memory size of the process.
SIZE	The virtual size of the process in (1024 byte) blocks.
SL	Sleep time the process has been blocked in seconds.
SSIZ	Size of the kernel stack for a given process.
STAT	Same as SV S. Displays the state of the process. The following table defines the possible status values. The status is given in four letters, for example, 'RWNA'.

Status	*Description*
letter 1 Indicates the run state of the process.	
D	The process is waiting on disk or is in some other short-term wait.
I	The process is idle, sleeping longer than 20 seconds.
P	The process is in page wait.
R	The process is runable. When CPU time becomes available the process will execute.
S	The process is sleeping for less than 20 seconds.
T	The process is stopped.
Z	Zombie processes. These processes have terminated and are waiting for their parent process to do a **wait(2)** call so they can exit.
letter 2 Indicates whether or not a process is swapped out.	
W	The process is swapped out.
>	The process has specified a soft limit on memory and has exceeded the limit. The process is not necessarily swapped out.
Blank	If the field is empty, the process is loaded in memory.
letter 3 Indicates whether or not a process is executing with altered CPU scheduling priority. The **nice** command changes scheduling priorities.	
N	The priority of the process has been reduced.
<	The priority of the process has been artificially raised.
Blank	If the field is empty, the process is running without special treatment.
letter 4 Indicates special treatment of the process for virtual memory replacement. The letters correspond to the options to the **vadvise(2)** system call.	
A	Usually represents a program which is performing garbage collection.
S	Usually represents a large image processing program that is using virtual memory to sequentially address large amounts of data.
TRS	The size of the resident set of text. The part of the process in real memory.
TSIZ	The Size of the text image. The text image is the shared program part of the process.
TT TTY	Same as SV TTY
UID	Same as SV UID
USER	Same as SV USER
WCHAN	Same as SV WCHAN

If a process shows <defunct>, the process has terminated and has a parent, but the parent has not yet waited on the child. The parent must perform a *wait* function to release the process.

PROCESSES A process or task is the existence of an executing program in the computer. Your login shell is a process while you are logged into the system. When you log off the system your login shell dies. The process is terminated or killed.

There is more than meets the eye when you refer to a UNIX process. A process includes an environment area, the status of files, additional memory allocations, user information, terminal device information, and the executable code. Processes are controlled by the UNIX kernel or operating system. The kernel assigns each process a unique number, called the process ID (PID). This allows the kernel and you to refer to the process on the same terms.

The kernel has to swap processes in and out of execution since multiple processes are using the same CPU. Thus the processes are always in a changing state. The **ps** command is used to display these various states during the life of a process.

DIAGNOSTICS AND BUGS **ps** (System V only) may not report the status of a process if you do not specify a *termlist*, *proclist*, *uidlist*, or a *grplist*. The reason is **ps** checks the *stdin*, *stdout*, and *stderr* to associate processes with your terminal. If you have redirected *stdin*, *stdout*, and *stderr*, there is no way for **ps** to associate the process with your terminal. Thus there will be no report for the process.

The **ps** command can only give an approximate report of process status since the information is continuously changing. You basically get a snapshot of the status of each process at the time **ps** retrieved the information. Therefore, the information you see is always a history and never actually current. On a lightly loaded system the information is less than a second old.

NOTE

If **ps** returns with no information and reports a message about "no namelist," the /etc/ps_data file may be missing. To rectify the problem create a ps_data file using **touch**. The date of the file must be set to a past time.

On a heavily loaded system it is possible for **ps** to report an *lseek* error and exit. This is caused by **ps** retrieving an address for a user's area and the user's process exiting before **ps** can *lseek* to the address. Thus when **ps** does *lseek* to the address it is no longer valid.

The **ps -ef** (System V only) command may report login times based on when the last **getty** was spawned for a tty instead of when the last user logged in to the system.

RELATED COMMANDS Refer to the **kill** and **nice** commands described in modules 70 and 94.

RELATED FILES The **ps** command uses many different files to retrieve the needed information. The following list describes each file used.

/dev/sxt*	Pseudo tty devices for (shl).
/dev/tty*	Terminal devices of all users (processes).
/dev/xt*	The terminal names being searched for by *ps*.
/dev/kmem	The system kernel's virtual memory.
/dev/mem	The system memory.
/etc/passwd	The system password file. *Ps* uses the UID information.
/etc/ps_data	The internal data structure used by *ps*.
/unix	The executable UNIX kernel. The namelist is needed.

APPLICATIONS

You use **ps** to monitor your background processes. It can also be used to see what other processes are executing on the system. You may have a process in background and decide you need to terminate it. Usually, you need to run **ps** to display the process ID of the process you wish to terminate. You can then use **kill** *PID* to terminate the process, where *PID* is the process ID from the **ps** listing.

TYPICAL OPERATION

In this activity you use the **ps** command to display all processes executing on the system. Begin at the shell prompt.

1. Type **ps -ef | pg** and press **Return**. Notice the owner of each process is listed in the first column.

BSD (Berkeley)

On BSD system the command is **ps -aux | more**.

Now list only processes related to your user_name.

2. Type **ps -fu mylogin** and press **Return**. The f option provides a full listing. The processes listed are related to your login shell.

BSD (Berkeley)

On BSD type **ps -u**.

---

3. Turn to Module 134 to continue the learning sequence.

Module 110
pushd (csh)

DESCRIPTION

The internal **pushd** command pushes directories around on the directory stack. The directory stack contains a list of directories you can change by using the **pushd** command. Each time a **pushd** command is executed, the shell **cd**s (changes directory) to the top directory entry on the stack. The stack entries are numbered from the top to the bottom starting at one. When the stack is displayed (after each **pushd**) the top is on the left.

If no argument is given, the top two directories on the stack are exchanged. The new top directory becomes the new present working directory.

If a new directory path is provided as an argument, it is pushed onto the top of the directory stack. The present working directory is changed to the new top directory.

If a *+n* is given as an argument, the stack is rotated so the *n*th entry becomes the top entry. For example, if the directory stack looked like this

```
/u1/ts/mylogin/d1 /u1/ts/mylogin/d2 /u1/ts/mylogin/d3 /u1/ts/mylogin/d4
```

and you typed

```
pushd +2
```

the stack would change to

```
/u1/ts/mylogin/d3 /u1/ts/mylogin/d4 /u1/ts/mylogin/d1 /u1/ts/mylogin/d2
```

and your present working directory would be /u1/ts/mylogin/d3.

COMMAND FORMAT The format of the **pushd** command follows.

```
pushd
pushd [ +n ]
pushd [ directory ]
```

Options Only one option may be used to control how **pushd** functions.

+n Rotate the *n*th entry in the directory stack to the top of the stack. All entries preceding the specified entry are moved to the bottom of the stack. The top stack entry becomes the new present working directory.

Arguments Only one argument may be passed to the **pushd** command.

directory The directory path to add to the top of the directory stack. The shell then changes directory to this new entry.

If no arguments are given, the top two directories on the directory stack are exchanged and the shell changes directories to the new top directory.

NOTE

The ksh implements **pushd** with functions. Check with your system administrator to see if pushd is available.

TIP: Most users of the **csh** alias the **pushd**, **popd**, and **dirs** commands. For example, you might want to add the following lines to your .cshrc file.

```
alias u pushd   # use u instead of pushd
alias o popd    # use o instead of popd
alias d dirs    # use d instead of dirs
```

RELATED COMMANDS Refer to the **dirs** and **popd** commands described in modules 36 and 104.

APPLICATIONS

The **pushd** command is useful when you are changing directories often. If you **cd** from one directory to another and back again, **pushd** can alleviate a large amount of typing. You simply type **u** *directory* instead of **cd** *directory* when you want to change directories. Then when you want to return to a previous directory you use the **dirs** command to display the directory stack. Select the desired directory and then type **u** *+n* to change directories to it.

TYPICAL OPERATION

In this activity you use the **pushd** command to create a directory stack and change directories. Begin at the shell prompt.

1. Type **pushd /usr/lib** and press **Return**. You are now in the system's lib directory.
2. Type **pushd /usr/spool/uucppublic** and press **Return**. Notice you have changed directory to the uucppublic directory. Try the pwd command to check.
3. Type **pushd ~** and press **Return**. Now you are in your HOME directory.
4. To return to the lib directory simply type **pushd +2** and press **Return**.
5. To return to the uucppublic directory type **pushd +3** and press **Return**.
6. Type **pushd +2** and press **Return** to return to your HOME directory.
7. Turn to Module 104 to continue the learning sequence.

Module 111
pwd

DESCRIPTION

The **pwd** command displays the full path of your present working directory. The *present working directory* is often referred to as the *current working directory* or *current directory*. Since the command that returns the present working directory is "pwd," we will refer to it as such. The **csh** uses the "cwd" variable to store the name of the current working directory. But it uses the **pwd** command to display the contents of the variable.

COMMAND FORMAT The format of the **pwd** command follows.

```
pwd
```

FURTHER DISCUSSION The UNIX System has file systems that are hierarchical tree structures containing directories and files. The system administrator created a directory somewhere in this hierarchy. When you log in you are positioned at this directory, known as your login or HOME directory. Since this is your home directory, you own it and all the files under it. Therefore, you have full permissions to read, write, create, change, or delete the contents of your home directory.

The **pwd** command is an alias command and an external command. The alias for **pwd** is "**print - $PWD**." The external command is located in /bin/pwd. Both of these commands perform the same function. The main difference is the internal **pwd** command executes much faster than the /bin/pwd command.

RELATED COMMANDS Refer to the **cd** command described in Module 14.

RELATED FILES The **pwd** command sends its output to the standard output.

APPLICATIONS

The **pwd** command is used to display where you are currently located in the file system hierarchy. It is used as a check for the cd command. You may not remember which directory you are in, and **pwd** will tell you. Once you **cd** to a new directory you use **pwd** to show the new path.

TYPICAL OPERATION

In this activity you use the **pwd** command to display your HOME directory and then, using cd to move, you display your parent directory. Begin at the shell prompt.

1. Display your present working directory by typing **pwd** and pressing **Return**.

```
cj> pwd
/u1/ts/mylogin
```

2. To change the present working directory type **cd ..** and press **Return**. Notice that no response is given. If the **cd** command worked, the directory change was successful.

3. Display the present working directory by typing **pwd** and pressing **Return**. Notice you have moved up the directory path one position. The ".." refers to the previous or "parent" directory in the present working path.

```
cj> pwd
/u1/ts
```

4. To return to your home directory type **cd** and press **Return**.

5. Turn to Module 110 to continue the learning sequence.

Module 112
read (sh, ksh), readonly (ksh) / set $<(csh)

DESCRIPTION

The internal **read** command reads one line from the standard input. Since it is an internal command it assigns the input to variables. The **readonly** command is used to set variables to readonly mode. If a variable is readonly, you cannot change it after the **readonly** command has assigned it a value.

COMMAND FORMAT Following is the general format of the **read** and **readonly** commands.

```
read [ var1 var2 ... ]
readonly [ var1 var2 ... ]
```

The **readonly** command assigns values to the listed variables and then sets them to **readonly** mode so the values cannot be changed later in the shell.

Csh Shell

```
set var=$<
```

Arguments The following arguments may be passed to the **read** command.

var1 The first variable in the list of variables. The first word on the input line is assigned to the first variable. Each input word is assigned to the corresponding variable on the **read** command line. All remaining input words are assigned to the last variable.

If no variables are given the input is discarded.

Csh Shell

var The variable *var* is assigned the contents of the line read from standard input.

FURTHER DISCUSSION The **read** command assigns each word of the input line to separate variables if they exist. The first word is assigned to the first variable, the second word to the second variable, and so on until all words are assigned. The last variable is assigned all remaining words on the input line. For example, if you typed

```
cj> read ONE TWO THREE
```

and then typed

```
cj> This line has too many words for the available variables.
```

the assignments of the three variables would be as follows:

```
cj> echo $ONE
This
cj> echo $TWO
line
cj> echo $THREE
has too many words for the available variables.
```

RELATED COMMANDS Refer to the **line** command described in Module 74.

RELATED FILES The **read** command reads from the standard input.

RETURN CODES A return code of 1 is returned if an EOF(Ctrl-D) is encountered. If the read completes, then a zero value is returned.

APPLICATIONS

The **read** command is used to read input from the standard input, usually your keyboard. It is primarily used for interactive shell scripts. If you need to ask questions or input data, you can use the **read** command to read the data from the keyboard. For example, the following lines of code might be used in a shell script for a user to enter his/her name.

```
cj> echo "Please enter your name(Last First MI): \c"
read LAST FIRST MIDINIT
```

This assigns the first word of the input to the LAST variable, the second word of input to the FIRST variable, and the remainder of the input to MIDINIT.

TYPICAL OPERATION

In this activity you use the **read** command to read lines from a file that has been redirected as standard input. Begin at the shell prompt.

1. Type **read LINE < /etc/passwd** and press **Return**. This reads the first line of the /etc/passwd file into the variable LINE.

C Shell

The **csh** has no internal way to read a line from a file. Here are some possible alternatives.

```
set LINE=`head -1 /etc/passwd`
set LINE=`line < /etc/passwd`
set LINE=`sed 1q /etc/passwd`
```

---

2. To display the LINE variable type **echo $LINE** and press **Return**.
3. To read from your keyboard type **read ANS** and press **Return**.

C Shell

3. To read from your keyboard type **set ANS=$<** and press **Return**.

4. Type **yes** and press **Return**.
5. Type **echo $ANS** and press **Return** to display the value of the ANS variable.
6. Turn to Module 42 to continue the learning sequence.

Module 113
Regular Expressions (REs)

DESCRIPTION

Regular expressions are patterns consisting of regular ASCII characters. They are used by various UNIX commands to search for a string of text that matches a given pattern. Most characters used in a regular expression pattern represent themselves. Special characters provide a way to restrict or generalize the pattern you are trying to match.

The **nawk, ed, egrep, ex, expr, grep, pg, sed, ksh,** and **vi** commands all use some form of regular expressions to perform these search and match functions. Unfortunately, each command has its own set of regular expressions. Thus there is no standard set to remember, although most of them share the same base of regular expressions with different additional regular expressions.

This module describes all possible regular expressions in full. Examples are provided for each pattern. Each command that utilizes regular expressions contains a brief listing of the regular expressions it supports.

REGULAR EXPRESSIONS The following table contains each regular expression and the task that it performs when used inside a pattern. Regular expressions are often referred to as REs in UNIX terminology. Thus we use the RE notation for uniformity and briefness.

Metacharacters Metacharacters are the special characters used in regular expression patterns that have special meanings. Metacharacters are often referred to as *special* or *magic* characters. The following is a brief listing of these characters.

\	Escapes the meaning of a metacharacter.
^	Matches the beginning of the line.
$	Matches the end of the line.
.	Matches any single character.
[ ]	A character class. Matches any one character in the class.
|	Alternation of regular expressions. Matches either one or the other of the regular expressions provided.
( )	Concatenation operation. Normally the parentheses are omitted.
*	Matches zero or more of the preceding regular expression.
+	Matches one or more of the preceding regular expression.
?	Matches zero or one of the preceding regular expression.

To have a metacharacter interpreted as a normal character, precede it with a backslash (\). Multiple metacharacters may be combined to define variable expressions consisting of only metacharacters.

BASIC REGULAR EXPRESSIONS Basic regular expressions make up the building blocks of all regular expressions.

Simple Regular Expressions Simple regular expressions are plain text that do not contain any metacharacters. Some examples follow.

NOTE

The UNIX operating system and its utilities are case sensitive. Each ASCII character is recognized by its unique code. Thus when you search for strings of text you must be aware of the uppercase verses lowercase mismatch. Regular expressions provide a means of circumventing the problems associated with case sensitive searches. There are also commands that can be used to translate text to uppercase or lowercase only.

John	Matches the word *John*. It does not match john or JOHN.
Go home now	Matches the string *Go home now*.
Not again	Matches the string *Not again*.

Single-character Regular Expressions Single-character regular expression are used to match themselves or various combinations of strings. They allow you to generalize the string you want to match. In some situations you may consider them as a way to restrict the matching of unwanted strings.

Character	*Description*
c	A single ordinary character that matches itself. For example: `ice cream` matches the string *ice cream*. Each character in the string has to match.
c	A single special character that has been escaped. It no longer has a special meaning and thus matches itself. Special characters are . (period), * (asterisk), [(left bracket), \ (backslash), ^ (caret), $ (dollar sign) and ~ (tilde). For example: `Tim\.` matches the string *Tim*.
	Match any single character except for a new-line. For example: `t.m` matches such strings as *tammie*, *timmy*, *tom*, *tommy*, and *tim*.
[*list*]	Matches any one of the ASCII characters listed within the brackets. The four special characters ., *, [, and \ represent themselves if used within the brackets. For example: `s[ea]t` matches *sat* and *set* but does not match *seat*.
[*c1-c2*]	Match any one of the ASCII characters in the range defined within the brackets. The - (hyphen) may be used to indicate a range of ASCII characters. This is part of the *list* feature described for the previous notation. For instance: `[0-9]` matches the range of number from 0 to 9. It is equivalent to [0123456789].

[^*list*] Do NOT match any of the ASCII characters listed within the brackets. Ranges may be specified. For example:

```
[^0-9]
```

matches all characters that do not contain a number.

Matches zero or more occurrences of the preceding regular expression. Matches the longest possible string. For example:

```
se*d
```

matches *sd*, *sed*, and *seed*. The pattern

```
s.*d
```

matches *sd*, *sed*, *seed*, *sid*, *sod*, *so don't move ahead* and various other strings. Remember, this regular expression matches the longest possible string. For example, if you have the following text:

```
Start at the starting line when we start.
```

and you use a regular expression like

```
s/t.*t/top/
```

you substitute stop for the longest possible string. The result is:

```
Stop.
```

Regular Expression Anchors Anchors are used in regular expressions to force the pattern to be considered only if it appears at the beginning or end of the line.

^ Matches the beginning of the line. The ^ (caret) must be the first character of the pattern. If it is not the first character, then it is considered an ordinary character. For example:

```
^Power tie
```

matches a line beginning with *Power tie*.

$ Matches the end of the line. The $ (dollar sign) must be the last character of the pattern. If it is not the last character, then it is considered an ordinary character. For example:

```
the end\.$
```

matches a line that ends with the string *the end*.

Regular Expression Word Anchors These regular expressions are only available in the **ex/vi** editor. They match the beginning or end of a word.

\< Matches the beginning of a word. A word is a string consisting only of characters, digits, and underscores. For example:

```
\<ram
```

matches *ram* and *ramming*, but does not match prog*ram*ming.

\> Matches the end of a word. For example:

```
the\>
```

matches *the* but does not match *these*.

Grouping Regular Expressions You can group regular expressions so you can reference them later in the same pattern or in the replacement string of a substitute command. These regular expressions are used by the **ed**, **ex**, **sed**, and **grep** commands.

\(RE\) Group the regular expression RE for later reference; each RE enclosed in parentheses is referenced by a corresponding number (see \n). The first group is referenced by \1, the second group by \2, and so on. You can have a maximum of nine groups in one regular expression pattern. For example:

```
\(dimmer\) \(light\) from the \2 \1
```

matches *dimmer light from the light dimmer*. A special case for grouping is the ability to find two occurrences of the same string on a line. For example:

```
^\(.*\)\1$
```

matches a string such as *the the* or any other string that is repeated on a single line.

\n Matches the *n*th grouped \(RE\) within the same regular expression. For example:

```
\(dog\) \(and\) \(cat\), \3 \2 \1
```

matches the string *dog and cat, cat and dog*. The \1 refers to the first string *dog*, the \2 refers to the second string **and**, and \3 to the third string *cat*.

Precise Occurrence Regular Expressions These regular expressions allow you to match exact numbers of a specified regular expression. These regular expressions are recognized by **ed**, **grep**, and **sed**.

\{*m*\} Matches exactly *m* occurrences of the preceding one-character RE. For example:

```
-\{5\}
```

matches ----- (exactly five hyphens).

\{*m*,\} Matches *m* or more occurrences of the preceding one-character RE. For example:

```
=\{13,\}
```

matches thirteen or more consecutive equal signs.

\{*m*,n\} Matches *m* through *n* occurrences of the preceding one-character RE. For example:

```
z\{2,3\}
```

matches *zz* and *zzz*. It does not match *z* or *zzzz*.

FULL REGULAR EXPRESSIONS Full regular expressions are an extended set of the basic regular expressions, with the exception of the grouping expressions \(RE\). The full regular expressions are only utilized by the **nawk** and **egrep** commands.

+ Matches one or more occurrences of the preceding regular expression. For example:

```
ba+d
```

matches *bad*, *baad*, and so on but does not match *bd*.

? Matches zero or one occurrence of the preceding regular expression. For example:

```
ba?d
```

matches *bd* and *bad* but not *baad*.

| or New-line Matches either one of the regular expressions separated by the | (vertical bar) or a new-line. For example:

```
I am|you are
```

matches the string *I am* or the string *you are*. The spaces have higher precedence than the |, thus the strings bind together instead of the two words next to the |.

( ) Group a regular expression for precedence control. By using the parentheses you can force the *, +, ?, and | metacharacters to apply to an entire regular expression. For example, consider the following two regular expressions:

```
John|Paul Jones
(John|Paul) Jones
```

The first regular expression matches *John* or *Paul Jones*. The second line matches *John Jones* or *Paul Jones*.

SPECIAL REGULAR EXPRESSIONS These regular expressions are used by the **ex/vi** editor.

Empty (Null) Regular Expression

// The last RE is used if it has been defined. For example:

```
/John/
//
```

The first line searches for the specified pattern *John*. The second line repeats the search for *John*.

Special Substitute Characters There are special characters that you can use in the replacement (new) string of a substitute command.

Special Character	*Description*

&	Matches last search or substitution or old string. For example,
	`The born is doing fine.`
	`:s/born/new&/p`
	`The newborn is doing fine.`
	or
	`My name is Tom.`
	`:s/Tom/&my/p`
	`My name is Tommy.`
	The first command substitutes *born* to be *newborn*. The second substitutes *Tom* to be *Tommy*.

~	Matches last replacement or new string. For example:
	`:s/finish/end/`
	`:s/stop/~/`
	The first line substitutes *end* in place of *finish*. The second command substitutes the last replacement string *end* in place of *stop*.

\l	Converts the next character in the replacement string to lowercase. For example,
	`:s/A big/\l&/p`
	`a big`
	The substitute command replaces the string *A big* with the string *a big*.

\L	Converts all following characters of the replacement string to lowercase until a \e or \E is encountered. Example:
	`:s/THIS/\L&\E/p`
	`this`

\u	Converts the next character in the replacement string to uppercase.
\U	Converts all following characters of the replacement string to uppercase until a \e or \E is encountered.

n	The *n*th grouped regular expression \(RE\). For example,
	`this is easy.`
	`:s/\(this\) is \(easy\)/\2 is \1/`
	`easy is this.`

\E or \e	Turn off the action of the \U or \L strings.

Multiple regular expressions may be used in a pattern. Some versions of **vi** do not support all the features of regular expressions shown here.

ESCAPE SEQUENCES The following sequences of characters are referred to as *escape sequences*. Escape sequences specify a notation for characters that are not easily produced from

your keyboard and/or cause problems when displayed to your screen. The **nawk** and **echo** commands use escape sequences. Although only **nawk** uses the escape sequences in regular expressions.

Sequence	*Description*
\b	Generates a backspace.
\f	Generates a form feed.
\n	Generates a carriage return.
\r	Generates a new-line.
\t	Generates a tab.
\0ddd	Generates the ASCII character for the given octal ASCII code.
\c	Generates the literal character *c*.

CONCATENATION OF REGULAR EXPRESSIONS Regular expressions can be placed beside each other to form more complex regular expressions. The following is a list of examples that describe some of the ways you can concatenate regular expressions.

.*	Matches the longest possible string of characters.
t.*t	Matches the longest possible string between two *t*s. For example: `So this is the end of that.` matches *this is the end of that* but not *this is t*.
t[^t]*t	Matches the shortest possible string beginning and ending with a *t*. The [^t] means not to match a t. In the previous example *this is t* is matched.
^\.	Matches any line beginning with a period.
^[^*Tab*]	Matches the beginning of a line that does not begin with a tab.
^$	Matches a blank line containing no characters. Match beginning of line and end of line.
^ *$	Matches a blank line containing only blank characters.
[0-9a-zA-Z]	Matches a single digit, lowercase character, or uppercase character.
[0-9]\{9\}	Matches any nine consecutive digits.
[01]\{8\}	Matches any eight consecutive ones or zeros.
[Tt]he	Matches *The* or *the*.
(bee)+	Matches one or more occurrences of the string *bee*. It matches *bee*, *beebee*, and so on but does not match *be* or *beee*.
(be?)+	Matches one or more occurrences of the string *be?*. Thus possible matches are *be*, *bee*, *bebe*, *beebee*, or *bebee*.
(I)|(Me)	Matches either " *I* " or " *Me* ."

APPLICATIONS

Regular expressions provide a shorthand for matching desired text patterns. Instead of having to type 20 different strings to search for in a file, you might be able to use regular expressions and only enter two or three regular expression strings.

You may also find regular expressions useful for defining strings you do not want to have selected. So they are helpful in selecting and rejecting various strings of text.

Regular expressions are used by the **nawk**, **ed**, **egrep**, **ex**, **expr**, **grep**, **pg**, **sed**, **sh**, and **vi** commands. By using regular expressions you can increase the power and effectiveness of these UNIX commands, thus increasing your productivity by better utilizing the features offered by each command.

TYPICAL OPERATIONS

In this activity you use the **egrep** and the **ksh** to utilize regular expressions. Begin at the shell prompt.

1. Type **egrep "mylogin|root" /etc/passwd** and press **Return**. Notice the two lines containing *mylogin* and *root* are displayed on your screen.
2. Type **egrep "^Some" db/phone** and press **Return**. All lines beginning with Some are displayed

   ```
   $ egrep "^Some" db/phone
   Someone: 916 Hawk Street::Austin:Texas:78749
   ```

3. Type **egrep "^[N]" db/phone** and press **Return**. All lines that do not begin with a capital N are displayed.
4. Type **egrep ".*" /etc/passwd** and press **Return**. Notice every line of the passwd file is displayed on your screen. The .* RE matches any number of any character.
5. In contrast, type **ls .*** and press **Return** to see how the shell uses the .* string. Notice only files beginning with a period are listed. The shell uses regular expressions differently than most other UNIX commands.
6. Turn to Module 77 to continue the learning sequence.

Module 114
rm

DESCRIPTION

The external **rm** command removes (deletes) files or directories. The remove command actually removes the specified links to a file. Since a file can have more than one link, performing a remove may not delete the contents of the file. Instead only the reference to the file is lost. Another reference may still exist. If all links (filenames) to the file are removed, the file is deleted and the contents are lost. You cannot undelete a file on a UNIX System.

COMMAND FORMAT The formats of the **rm** command follow.

```
rm [ -fi ] file_list
rm [ -fir ] directory_list [ file_list ]
```

BSD (Berkeley)

```
rm [ -fi ] [ - ] file_list
rm [ -fir ] [ - ] directory_list [ file_list ]
```

Options The following list describes the options used to control how **rm** functions.

-f Forces the removal of all files listed. Does not check to see if the file is writeprotected or not. Does not prompt for confirmation. If the directory is writeprotected the files are NEVER removed.

-i Interactive file removal. Before each file is removed you are requested to respond with a confirmation. Any string beginning with a "y" removes the file. If used with the -r option, confirmation is asked for each directory. The -i option overrides the -f option and is effective even if the standard input is not your terminal keyboard.

-r Recursively removes files and directories. All files are removed from each directory. Each directory is then removed. If a file is writeprotected (you do not have write permission) then **rm** prompts you for confirmation to remove the file.

BSD (Berkeley)

- Informs **rm** that the following arguments are filenames, not options. This allows you to remove files beginning with a - that would normally be interpreted as invalid options to remove.

Arguments The following list describes the arguments that may be passed to the **rm** command.

directory_list One or more directories to be searched for files to remove. If the -r option is used, all files under the listed directory are removed and the directories themselves are deleted.

file_list One or more files unlinked from the inode and removed from the directory file.

FURTHER DISCUSSION The **rm** command removes the entries for one or more files from a directory, thus removing a link to the inode. If the last link pointing to the inode is removed, then the contents of the file can no longer be referenced. The inode and contents are still on the disk but you have no way to access them.

NOTE

The system administrator may be able to use the **fsdb** program to relink the file if it is important and you do not have a backup. But don't bank on this possibility. Most system administrators are not familiar with **fsdb**. Plus, the file system has to be umounted immediately after the file is removed. The next file allocation request may use the inode you just freed, leaving absolutely no record that the file ever existed.

You do not have to have read or write permissions on a file to remove it. Although you do have to have write permission for the directory to remove a file from the directory. If you do not have write permissions for the file and you are using **rm** from your terminal, **rm** prompts with the file's permissions and waits for a confirmation. To perform the removal, type any string beginning with a "y."

The -r command must be present for **rm** to remove a directory.

TIP: If you are not sure how the shell expands an ambiguous file reference, use the **echo** command to evaluate which files are referenced. Then use rm -i to remove the desired files.

DIAGNOSTICS AND BUGS The **rm** command will not allow you to remove your current directory (.) or your parent directory (..). For example, the following command

```
rm -r .*
```

will not remove your present working directory or your parent directory.

RELATED COMMANDS Refer to the **chmod**, **ln**, and **rmdir** commands described in modules 17, 76, and 115, respectively.

RETURN CODES The **rm** command returns an exit code of zero if all files and directories are removed successfully. If any file or directory is not removed, then a nonzero status is returned.

APPLICATIONS

The **rm** command is used to remove unwanted files. Old versions or copies of files can be discarded. Multiple links to a single file can be reduced to only one link for the file. It can be used to clean up entire directory tree structures after the structure has been moved to a new location or saved to tape. You will use **rm** repeatedly to clean up disk space.

CAUTION

Be sure you reference only the files you no longer need. Once you remove a file on UNIX it is gone forever. The only way to recover it is from a tape or floppy backup!

TYPICAL OPERATION

In this activity you use the **rm** command to remove a file in a subdirectory but leave the directory intact. Begin at the shell prompt.

1. Type **rm stuff** and press **Return**. The file "stuff" is removed from your present working directory.
2. Type **rm -ri letters** and **Return**. Notice you are prompted for confirmation to remove the file.

```
cj> rm -ri letters
letters/calendar? y
letters/file1? y
letter?
```

3. At the prompt to remove the *letters/calendar* and the *letters/file1* files, type **y** and press **Return**. Press **Return** to leave the *letters* directory intact.
4. Turn to Module 138 to continue the learning sequence.

Module 115
rmdir

DESCRIPTION

The external **rmdir** command removes a directory. The directory must be empty. An empty directory only has two entries, the . (dot) representing itself and the .. (dot dot) representing its parent directory.

The **rmdir** command is a safer way to perform directory removal than using the **rm -r** command. It checks the directory for the existence of files and subdirectories. If any exist, **rmdir** complains and exits without removing the directory.

NOTE

It is invalid to remove your current "." (dot) directory or your parent ".." (dot dot) directory.

COMMAND FORMAT Following is the general format of the **rmdir** command.

```
rmdir [ -ps ] directory_list

rm -r directory_list
```

Options The following options may be used to control how **rmdir** functions.

-p Removes empty directories specified in the directory path. If you specify a directory by a pathname and its parent directories are empty, they also are removed. The name of each directory is displayed on the standard output as it is removed. Directories that are not removed are displayed on the standard error.

-s Suppress any messages produced. Both standard error and standard output are suppressed.

BSD (Berkeley)

Berkeley does not support any options on the **rmdir** command.

Arguments The following argument may be passed to the **rmdir** command.

directory_list One or more directories you wish to remove. The directories must be empty. The directory names used can be single filenames or pathnames.

DIAGNOSTICS AND BUGS The following messages may be returned by **rmdir**.

nonempty directory:	The directory contains files. Remove the files by typing **rm directory_name/*** or use the **rm -r directory_name** command.
current directory illegal	You cannot remove your current or parent directory. **cd** to a different directory and then remove the directory. You must be in a directory at least one level above the one you wish to remove.

RELATED COMMANDS Refer to the **mkdir, rm, ksh,** and **umask** commands described in modules 89, 114, 71, and 147 respectively.

APPLICATIONS

The **rmdir** command is useful for removing a directory. It is the restrictive or safe way to remove a directory. It will complain if any files exist in the directory and fail. You have to remove all files in the directory before **rmdir** will remove the directory. The **rm -r** command is a much more powerful command, but with power comes the hazards of carelessness. UNIX has no mercy when it comes to deleted files. Therefore, be extremely careful if you venture to use the **rm -r** command.

TYPICAL OPERATION

In this activity you use the **rmdir** command to remove a directory you created by mistake. Begin at the shell prompt.

1. Type **rmdir misc/misc misc** to remove the unwanted directories named *misc/misc* and *misc*.
2. Type **ls -l** and press **Return** to display the absence of the *misc* directory.
3. Turn to Module 18 (SV), Module 16 (BSD) to continue the learning sequence.

Module 116
script

DESCRIPTION

The external **script** command makes a copy (typescript) of a terminal session. When you run **script** a new shell is forked. This new shell makes a complete copy of everything displayed on your terminal. It reads standard input and output for your terminal tty and stores the data in a *file*. The default filename is *typescript*.

To exit from a script session you simply press Ctrl-D or type **exit**.

COMMAND FORMAT Following is the general format of the **script** command.

```
script [ -a ] [ file ]
```

Options The following option may be used to control how **script** functions.

-a — Append the output of **script** to *file*. Normally **script** begins writing to a new file; if the file exists it is overwritten unless you specify the -a option.

Arguments The following argument may be passed to the **script** command.

file — The *file* script uses to save all data.
If no *file* is specified then the file named *typescript* is used.

RELATED COMMANDS Refer to the **tee** command described in Module 134.

RELATED FILES If no output file is specified, the output of **script** is placed in the *typescript* file.

APPLICATIONS

The most common use of **script** is to make a copy of a terminal session. By starting **script** you can save all the information, displayed on your terminal during a login session, to a file. You can then print the file or view it with an editor.

NOTE

Remember, if you used applications with cursor control, those control characters will reside in the output *file* produced by **script**. Therefore, if you send the *file* to a printer or terminal, it may not print or display properly. You can use the **col** command to remove control characters from the typescript file.

TYPICAL OPERATION

In this activity you use the **script** command to make a typescript of a few commands. Begin at the shell prompt.

1. Type **script** and press **Return**. Your output is saved in the *typescript* file.

```
cj> script
Script started, file is typescript
cj>
```

2. Now perform several UNIX commands. For example, type **ls** and press **Return**.
3. Type **cat /etc/group** and press **Return** to have output sent from the computer to your terminal.
4. Now stop **script** from tracing. Type **exit** and press **Return**.

```
cj> exit
Script done, file is typescript
cj>
```

5. Display the output of **script** by typing **more typescript** and pressing **Return**.
6. Type **rm typescript** and press **Return** to remove the output file created by **script**.
7. Turn to Module 60 to continue the learning sequence.

Module 117
sed

DESCRIPTION

The external **sed** command is a stream editor. A stream editor reads input from a list of files or from the standard input, applies editing commands to each line of input and writes the resulting line to the standard output. The standard output by default is your terminal screen. The input data is often referred to as a stream (of data), thus the name "stream editor."

Some of the editing commands are derived from the **ed** editor. These commands can be specified on the command line or within a file. They can have addresses specified as line numbers or as patterns. The editing commands are referred to as the sed *script*.

COMMAND FORMAT Following is the general format of the **sed** command.

```
sed [ -n ] [ -e script ] [ -f scriptfile ] [ file_list ]
```

Options The following list describes the options and their arguments that may be used to control how **sed** functions.

-n — Suppresses the default action of writing every input line to the standard output.

-e *script* — The *script* is a set of editing functions used to edit the input data. If you only specify one set of *scripts* and you do not specify the -f option, you may omit the -e. You usually have to enclose the *scripts* in single or double quotes to prevent interpretation by the shell. You can use one -e option and place multiple edit commands within quotes or you can use multiple sets of -e *scripts* to perform the edits. For example, using a single -e with multiple edit commands in a single *script* might look like the following:

```
sed -e 's/BEGIN/START/g
       s/NIGHT/DAY/g
       /^$/d' inputfile > outputfile
```

This applies each edit command (function) to each input line. The same effect can be accomplished using multiple -e options. For example,

```
sed -e 's/BEGIN/START/g' -e 's/NIGHT/DAY/g' \
   -e '/^$/d' inputfile > outputfile
```

-f *scriptfile* — Allows you to store an edit *script* in the file *scriptfile*. You can use multiple -f *scriptfile* options to have **sed** use more than one *scriptfile*. The edit functions are read from the *scriptfile* and applied to the input data stream.

Arguments The following describes the argument that may be passed to the **sed** command.

file_list — The list of input files containing the data for editing.

FUNCTIONAL OVERVIEW The **sed** command processes one line of input at a time. It reads in a line and stores it in the temporary work buffer referred to as the *pattern space*. The address of each editing command is compared to the address or text of the pattern space for a match. If a match occurs, the edit command is applied to the text stored in the pattern space.

All edit commands in all **sed** scripts are applied to the text stored in the pattern space. Once all edit commands have been applied to the pattern space text, the pattern space is copied to the standard output and then deleted. The next line of input is read into the pattern space and the process is repeated. Thus **sed** processes each line of input in a cycle, reading input, performing all editing commands that apply, writing to the output, and clearing out the work area.

The text stored in the pattern space can be copied to a holding buffer called the *hold space*. The text in the hold space can later be copied back to the pattern space in part or whole. There are three edit commands that store text in the hold space; they are the h, H, and x commands.

ADDRESSES The **sed** command supports four types of addresses. These addresses are used to match the text in the pattern space. If a match occurs based on the given address, the edit commands following the address are applied to the pattern space. The four types of addresses are:

Address	*Description*
no address	If no address is given, the editing command is applied to every pattern space (input line).
number	If a number is given for an address, the editing command is applied to the pattern space whose input line number is equal to the given address number. If multiple input files are listed, the input lines are numbered consecutively from one file to the next. Therefore, there is only one line number 1.
$	The dollar sign signifies the last line of input. If more than one file is given, then $ refers to the last line of the last file.
context	A context address is a regular expression supported by the **ed** command. The regular expression is placed between two delimiters; normally the delimiters are slashes. For example, `sed -e '/pie/s/Apple/Cherry/' fruits` searches for an input line that contains the pattern "pie." If "pie" is found, the pattern space is selected and all edit commands are applied. You can change the delimiters for context addresses if you wish. Start the address with \c, where c is the character you want to use as the delimiter. The next character c is used as the second delimiter. To use c within the regular expression you must escape its meaning with a backslash. For example, `sed -n '\+Two\+Two+p' inputfile` matches the pattern "Two+Two."

S1 S2
inter space [ICMP]
A B
may not be considered similar hence both proposal
$
"may be only Time Earth Quake"
26/04/2013

The context address may contain a new-line character. Unlike the **ed** pattern selection, the **sed** context pattern can contain the "\n" notation to match a new-line in the pattern space. Initially, the pattern space contains only one new-line which cannot be matched by \n. But you can use editing commands to add additional lines to the pattern space, thereby placing embedded new-lines in the pattern space.

A period (.) matches any character except the new-line.

Address Ranges Any of the four *types* of addresses can be combined to form an address range. An address range matches all input lines from the first address through the second address. For example,

```
sed -e '/Apple/,/Orange/s/Pie/Jam/' fruit
```

would match all lines from the first line containing "Apple" to the first line containing "Orange" and change "Pie" to "Jam."

Using numbers for address ranges might resemble

```
sed -e '5,10s/Pie/Jam/' fruit
```

which would change Pie to Jam on lines 5 through 10.

Using Addresses There are three *forms* of addressing used by editing commands. The forms are:

no address — If no address is given then every pattern space (input line) is selected.

single address — If a single address is given, then only pattern space text matching the given address is edited. For example, if you specify a line number, then only that input line is edited. If you specify a context address, then only those lines containing the regular expression are edited.

two addresses — If two addresses are specified, then all lines inclusive of the two addresses are edited. For example, if you specify

```
sed -e '1,5s/it/the dog/g' inputfile
```

lines 1 through 5 are edited. See the previous section describing Address Ranges.

EDITING COMMANDS The editing commands used by **sed** resemble those of the **ed** editor. Some of the commands are borrowed from **ed**, others are added for **sed**'s special use. The editing commands can be specified on the command line using the -e option or placed in files. These commands are referred to as **sed** *scripts*. Multiple scripts can be defined on the command line by enclosing them in quotes or by using the -e option repeatedly. More than one edit command can be placed in a scriptfile.

These editing commands can be split into three categories. Those that perform text manipulation on the pattern space, those that add new text to the output, and those that provide flow control.

Command Descriptions The following list describes the editing commands supported by **sed**. The addresses supported by each edit command varies. To show the maximum number of addresses a command supports, we use the [n] notation for the address, where n is the maximum number of addresses. For example,

No address means you cannot use an address with the edit command.

[1] One address can be used, for example,

```
/John/d
5d
```

The first command deletes all lines containing "John," the second deletes line 5.

[2] Two addresses may be used. The addresses must be separated with a comma. For example,

```
/John/,/Mark/d
10,25d
```

The first line deletes all lines from the first occurrence of "John" to the first occurrence of "Mark." The second line deletes lines 10 through 25.

Command	*Function*

[1]a\ *text*	The *a* command appends *text* to the output after the current pattern space is written but before reading the next line of input. The *text* is not placed in the pattern space, thus the *text* is not modified by later edit commands. For example, if you have the following input
	`This is line 1`
	`This is line 2`
	and the sed statement is
	`sed 'a\`
	`This is appended text.' input`
	the output would be
	`This is line 1.`
	`This is appended text.`
	`This is line 2.`
	`This is appended text.`

[2]b *label*	Branches (jumps) to the specified *:label* command. The label can be from one to eight characters in length. If no label is given, then you branch to the end of the sed script. This allows conditional editing scripts.
[2]c\ *text*	Deletes the contents of the pattern space and sends the *text* to the output. All remaining editing commands are skipped since the pattern space is empty.
[2]d	Deletes the contents of the pattern space. All remaining editing commands are skipped since the pattern space is empty.
[2]D	Deletes the segment of text preceding the first embedded new-line from the pattern space. Another command must first be used to place multiple lines in the pattern space. All remaining editing commands are skipped even though there is text in the pattern space. The pattern space is not displayed to the output. It can, however, be saved to the hold space. If no other command, such as N, has been used, then D functions like the d command.

[2]g — Replaces the contents of the pattern space with the contents of the hold space.

[2]G — Appends the contents of the hold space to the contents of the pattern space.

[2]h — Replaces the contents of the hold space with the contents of the pattern space.

[2]H — Appends the contents of the pattern space to the contents of the hold space.

[1]i\ *text* — Writes the *text* to the standard output before the current pattern space is written to the standard output. The *text* is not placed in the pattern space; therefore, it is not affected by the remaining editing commands. For example, if your input was the same as the input used for command a and the **sed** statement was

```
sed 'i\
This is inserted text.' input
```

the output would be

```
This is inserted text.
This is line 1.
This is inserted text.
This is line 2.
```

[2]l — Write (list) the contents of the pattern space on the standard output, substituting nonprintable characters (control characters) with two-digit ASCII codes. Lines longer than 80 columns are wrapped onto the next line. **sed** only supports ASCII character codes up to octal 177 (Delete); it does not support extended ASCII.

[2]n — Display the contents of the pattern space on the standard output. Replaces the pattern space with the next line of input. This allows you to read in lines of text and send them to the standard output without being edited by the other edit commands in the script.

[2]N — Appends the next input line to the pattern space. The two lines of text are separated with a new-line. The line number is increased by one.

[2]p — Writes (print) the pattern space on the standard output. By default **sed** copies the pattern space to the standard output. Thus when you specify the p edit command, each line is printed (displayed) on the standard output twice, unless you specify the -n option on the command line.

[2]P — Writes (Prints) the segment of text preceding the first new-line from the pattern space. The P command functions like the p command if a command such as N has not been used. The P command allows you to add multiple lines of input to the pattern space, display the first line, and then manipulate the pattern space as desired.

[1]q — Quit **sed**. Branches to the end of the script and stops processing input (exits sed).

[2]r *rfile* — Reads the entire contents of *rfile* and writes them to the standard output before reading the next line of input. The format of this command must be r followed by one space or one tab and the name of the *rfile*, with nothing after the filename.

[2]s/*RE*/*rs*/flags — Substitute the replacement string *rs* for the regular expression *RE*. See the regular expression section below for a list of supported REs. The / (slash) may be replaced by any other character. A flag may be zero or more of the following:

g	Substitutes *rs* for all occurrences of *RE* in the pattern space. A global line substitute instead of a first occurrence substitute. The substitution is nonoverlapping, which means *sed* scans for the first occurrence, performs the substitution, then searches for the next occurrence. For example, if the pattern space contained "meesseessippi," the command **s/e.s/is/g** would result in "mississippi" even though e.s matches ees and ess. The ess never gets changed because the ees changes to iss before the ess is located.
n	Substitute only the *n*th occurrence of the RE with *rs*. The value of n can be from 1 to 512.

BSD (Berkeley)

Some BSD versions do not support the n option of the substitute command.

---

p	Write (print) the contents of the pattern space to the standard output if a replacement was made.
w *wfile*	Append the contents of the pattern space to the file *wfile* if a replacement was made.
[2]t *label*	If a substitution has occurred since the last input line was read or the last *t* command was performed, then branch to the :*label* statement. If *label* is empty then branch to the end of the script.
[2]w *wfile*	Write (append) the pattern space to the file *wfile*. There can only be one space or tab between the w and the filename. You can have up to ten *wfiles* in a script.
[2]x	Exchanges the contents of the hold space with the contents of the pattern space. The h and H commands place text in the hold space.
[2]y/*str1*/*str2*/	Transforms each character in *str1* to the corresponding character in *str2*. For example, `y/abc/xyz/` changes every occurrence of an a to x, b to y, and c to z. The length of *str1* and *str2* must be identical.
[2]!*command*	Apply the edit *command* to input lines that do not match the specified address. For example, `sed -n '/Total/!p' inputfile` displays all lines that do not match "Total."
:*label*	The : (colon) signifies a label follows. The *label* provides a unique position in the script for the *b* and *t* commands to branch to, such as *goto* addresses.
[1]=	Writes the current input line number to the standard output on a line by itself.

[2]{
script
}

Execute the edit commands in *script* when an input line matches the specified address. The left brace ({) follows the given address, each command in *script* must be on a line by itself, and the right brace (}) must be on a line by itself. For example,

```
sed -n '/UNIX International/{
H
G
}
p' inputfile
```

This appends the pattern space to the hold space, then appends the hold space back to the pattern space. Thus each time the string address is matched, the number of times it is output increases. Useless, but an example nevertheless.

If a # (number sign) is the first character on the first line of a script file, then the entire line is ignored and can be used as a comment line. If the # is followed by an "n," #n, then all default output is suppressed. This is the same as using the -n option on the **sed** command line.

A blank line or empty command is ignored.

DIAGNOSTICS AND BUGS The error messages returned by **sed** are usually difficult to understand and usually provide little help in understanding the problem. Typically, you have left out a quote, slash, or edit function letter and **sed** does not understand the syntax. The only help is to turn to the list of editing commands to make sure you are using the correct command syntax and to double check the placement of quotes and substitute delimiters (usually slash marks).

RELATED COMMANDS Refer to the **nawk**, **ed**, and **grep** commands described in modules 6, 39, and 60.

RELATED FILES The **sed** command reads input from a list of files or the standard input and writes the edited results to the standard output.

APPLICATIONS

The **sed** command is a useful shell script tool. Because it is a stream oriented editor it can be used in pipe commands, reading input from one command and passing output to another. It is also a batch type command, not requiring interactive input from a user. Thus the edit commands are all predefined and are applied to the input data upon execution.

The most common uses of **sed** are usually in shell scripts where the easiest way you can think of performing data manipulation is by using an editor. If you feel using an editor is the way to perform the changes, then use **sed**. This brings us to the flexibility of UNIX. Most tasks can be resolved using various commands. On one hand this is good but on the other it may seem confusing at first.

TYPICAL OPERATION

In this activity you use the **sed** command to display the contents of a file, substitute strings, and insert new text. Begin at the shell prompt.

1. Type **sed -n p mgmt** and press **Return**. Notice that the file is displayed on your screen as if you had used the **cat mgmt** command.
2. Type **sed p mgmt** and press **Return**. Now each line is displayed twice.

```
cj> sed p mgmt
Andrew
Andrew
Barbara
Barbara
Kathy
Kathy
Scott
Scott
```

   The -n option suppresses the normal output of every line. The p command prints the contents of the pattern space in addition to the default printing of it.

3. Type **sed 's/$/ (Manager)/' mgmt** and press **Return** to change each occurrence of the end-of-line to " (Manager)."
4. To add lines to the file, you use the i command. Type the following **sed** command, editing commands, followed by the text.

C Shell

If you are using the **csh**, type **sh** and press **Return** to enter a Bourne shell for the following example.

```
cj> sed '3i\
> Bill\
> Leslie' mgmt
Andrew
Barbara
Bill
Leslie
Kathy
Scott
```

5. Press **Ctrl-D** to return to your **csh**.
6. Type **sed 2q mgmt** and press **Return** to display only the first two lines of the input file.
7. Turn to Module 39 to continue the learning sequence.

Module 118
set/unset

DESCRIPTION

The internal **set** command is used to control the current shell environment. It can perform three basic functions:

- Set options to control how the shell functions.
- Set positional parameters.
- List local environment; the name and value of variables.

C Shell

The C shell utilizes the **set** command differently from the Bourne or Korn shells. It assigns a value to a variable and displays the name and value of all variables.

The internal **unset** command removes variables and functions from the shell environment. It does not set them to NUL (\0) but removes the name from the variable table of the shell. Setting a variable to NUL leaves the variable name in the shell variable table with a value of NUL.

COMMAND FORMAT Following is the general format of the **set** command.

```
set [ -aefhkmnopstuvx ] [ -o opt_arg ] [ -A name ] [ args ]
set [ +aefhkmnopstuvx ] [ +o opt_arg ] [ +A name ] [ args ]

unset [ variables ]
unset [ -f ] [ functions ]
```

C Shell

```
set variable
set variable=value
set variable=variable[index]
set variable=(wordlist)
set variable=$<

unset variable
```

Options The following list describes the options and their arguments that may be used with the **set** command.

-- If arguments follow, do not interpret as options. Prevents a -opt in $1 from being interpreted as an option. If no arguments follow then **unset** all positional parameters.

-a Automatically export all variables as they are assigned values. Implies that an **export** ***variable*** is performed after each variable is assigned a value.

-e Exit the shell immediately if running a shell script and an error occurs.

-f Filename generation is disabled. The shell no longer expands wildcard characters.

-h Causes each command that you enter on the command line to become a tracked alias.

-k Place all keyword arguments in the environment for a command. By default, only the keyword arguments preceding the command are exported to the environment.

-m Enable the job monitor. A line will be displayed upon completion of each background job. The exit status of each background job is displayed with the completion message.

-n Read but do not execute commands. Useful to check for syntax errors in shell scripts.

-o *optargs* Allows you to set other and special options.

allexport	Same as -a
bgnice	Run all background jobs at a lower priority.
emacs	The *emacs* style in-line editor is used for command reentry. See In-line Command Editing in the **ksh** command, Module 71.
errexit	Same as -e
gmacs	The *gmacs* style in-line editor is used for command reentry.
ignoreeof	If the shell is the *login* shell, it will not exit on end-of-file (Ctrl-D). The **exit** command must be entered.
keyword	Same as -k
markdirs	A / is appended to all directory names resulting from pathname expansion.
mapesc *value*	Map the escape character (033) to a new value.
noclobber	Redirection will not overwrite an existing file. You must use the >| symbol to overwrite an existing file.
noexec	Same as -n
noglob	Same as -f
nolog	Function definitions are not stored in the history file.
nounset	Same as -u
privileged	Same as -p
protected	Same as -p, except PATH is reset.
verbose	Same as -v

	vi	The **vi** style in-line editor is used for command entry. See In-line Command Editing.
	viraw	Specifies **vi** character-by-character input.
	xtrace	Same as -x
-p	Toggles the privileged mode on and off. If on, the effective user and group IDs are set to the real ones. If off, the effective IDs are restored to their values when the shell was initially invoked. Thus privileged is on when the effective ID does not match a real ID. When privileged mode is on, the shell • Disables processing of the $HOME/.profile file. • Uses the /etc/suid_profile instead of your ENV file.	
-s	Sorts the positional parameters.	
-t	Read and execute one command then exit the shell.	
-u	Unset variables cause an error when substitution is performed. If a variable has not been assigned a value and it is referenced, then an error will occur.	
-v	Verbose; echo (display) each shell input line as read.	
-x	Debug; display command lines as they are executed, each command line is preceded by a +.	
-A	Assign arguments *arg* to array *name* in sequential order beginning with element 0.	
+opt	If any of the above options are preceded by a + (plus sign) instead of a - (hyphen), the option will be disabled or turned off.	

If no options or arguments are given, **set** displays the local environment.

NOTE

The **set** command can be used to change or set options and arguments once a shell has begun execution. Set cannot change or set the -c, -i, -r, and -s options from within the current shell.

The following list describes the options and their arguments that may be used to control how **unset** functions.

-f	The shell unsets only functions that you list on the command line. Variables are not affected.

Arguments The following list describes the arguments that may be passed to the **set** commands.

args	Valid arguments are programs or shell scripts that are executable and their arguments. These *args* are referred to as *positional parameters* (see Module 99, Parameters/Variables/Arrays). Parameters may be *keyword parameters, constants, tilde substitution, command substitution*, or *filename generation.*

Csh Shell

(*wordlist*)	A list of words that are assigned to the named variable. Command and filename expansion is performed. Variable expansion is also performed before the list of words are assigned to the variable. Multiple lists may be specified.

$<	Reads one line from standard input.

The following list describes the arguments that may be passed to the **unset** command.

variables Shell variables that have been created. They may or may not be set to a value, but they do exist in the environment.

functions Shell functions that have been defined (ksh only).

FURTHER DISCUSSION The **set** command provides three functions. You can display the names and values of all variables (parameters) by typing set with no options or arguments. You can set positional parameters by typing

```
cj> set positional_parameter1 positional_parameter2
```

and pressing Return. You may also want to use **set** to change the current options of your shell or shell script. For example, you may be writing a script and have a syntax error. To debug the script you can insert a line containing **set -x** to have each line displayed as it is executed. This is equivalent to typing **sh -x shell_script**.

The following examples help clarify some of these options.

```
cj> ksh -x myscript

cj> START=10 ksh -k myscript INCREMENT=2
```

The -x can be used to debug shell scripts. The -k option allows you to place keyword assignments after the command has been entered.

To display what options are currently set for the shell type

```
echo $-
```

and press Return.

The **unset** command will remove a variable or a function from your current shell environment. It will not remove a readonly variable. It cannot remove the PATH, PS1, PS2, MAILCHECK, or IFS variables.

RELATED COMMANDS Refer to the **ksh** command described in Module 71 and the **shift** command in Module 120.

APPLICATIONS

The **set** command may be used to set or reset options for the current shell, thus allowing you to change how the shell performs in-line, without having to restart your shell (log out and log in again).

You can also use **set** to set and reset positional parameters. This provides a simple way to work with columns of data separated by the character(s) defined in the IFS variable. You can **cat** a file to a loop structure, read one line per iteration, and use **set** to assign the line to positional parameters. Each word of input becomes a parameter.

The **unset** command is used to completely remove a variable or function from the shell environment. You may decide you no longer want a variable to exist in the current environment. If you are going to invoke a command or subshell, you may not want to pass your full environment

to the new child process. By unsetting the variables you prevent the new process from knowing about the variable.

TYPICAL OPERATION

In this activity you use the **set** command to process a line of input from a file inside a shell while loop. Begin at the shell prompt.

If you are using the **csh**, type **sh** and press **Return** to enter a Bourne shell for the following examples. Eliminate the "--" and escape sequences from the **echo** commands if present. Change the "\t" to tabs.

---

1. Type the following lines, pressing **Return** after each line.

```
cj> OIFS=${IFS}
IFS=":"
cat /db/phone | \
while read LINE
 do
   set -- ${LINE}
   echo "$1\t$4\t$5"
 done
IFS=${OIFS}
```

   This code will list out each line of the /db/phone file and format it into a readable table. The IFS=: changes your field (argument) separator to a colon. The while loop reads one line of input per iteration of the loop. When read can no longer read input (EOF is read) then loop terminates. The set -- ${LINE} sets the input line to be positional parameters. The backslash t (\t) tells echo to display a tab. You must reset your IFS variable back to normal so you can pass arguments correctly.

C Shell

1. There is no easy or reliable way to duplicate this code in the csh. But you can reset the command line arguments using the set command. Simply type the following.

```
cj> set argv=(newarg1 newarg2 newargN)
```

2. Now type **echo $argv[2]** and press **Return**.

---

2. The **set** command can list out your local environment. Type **set** and press **Return** to list the environment. In contrast to **set** displaying your local environment, you can use **env** or **printenv** (BSD) to display your global environment.

In this activity you use the **unset** command to remove a variable from your current environment. Begin at the shell prompt.

1. Type **MYVAR="Here is a new variable defined"** and press **Return**.

2. Display the value of MYVAR by typing **echo $MYVAR** and pressing **Return**. Your screen will look like this:

```
cj> echo $MYVAR
Here is a new variable defined
```

3. Type **set** and press **Return** to display all variables in your shell environment. Notice the MYVAR variable and its value.
4. Now remove the MYVAR variable by typing **unset MYVAR** and pressing **Return**.
5. Type **set** and press **Return** again to display your environment. Notice that the MYVAR variable is not listed. It no longer exists in your environment.
6. Turn to Module 119 to continue the learning sequence.

C Shell

The **set** command is used in the **csh** to set variables. The following activity illustrates the use of the **set** command.

1. Type **set term=unkown** and press **Return**.
2. Now type **set** and press **Return** to display all set variables.
3. Type **set term=vt100** and press **Return**. Replace *vt100* with the type of terminal you are using.
4. Turn to Module 119 to continue the learning sequence.

Module 119
setenv/unsetenv (csh)

DESCRIPTION

The internal **setenv** command sets an environment variable to a given value. This allows you to use a meaningful name to contain a value. The value can be changed as required. All variables are exported to subshells.

These variables are placed in the shell's environment. The shell maintains two sets of variables, the local variables and the global, or environment, variables. Local variables are only known to the current shell. Environment variables are known by the current shell and all processes started by the current shell.

The **unsetenv** command removes a variable from the environment. The value and the variable name are discarded.

COMMAND FORMAT Following is the general format of the **setenv** command.

```
setenv
setenv variable
setenv variable=value

unsetenv variable_pattern
```

Arguments The following list describes the arguments that may be passed to the **setenv** command.

variable	A word used to store a value.
value	The alpha, numeric, or alphanumeric value stored in a given variable.
variable_pattern	A pattern expanded by the shell which removes all variables that match it.
	If no argument is given all environment variables are displayed.

FURTHER DISCUSSION Exported variables exist in the environment space of the shell process. When the shell executes a new child process (subprocess) it passes the environment to the new process. The environment consists only of variables that have been exported. By using exported variables you can pass "global" information to any subprocess.

Subprocesses cannot change the environment of a parent process. They can change their environment but the changes cannot be passed back to the parent.

If **setenv** is executed without any arguments, on some systems, it lists out all exported variables; on others, it returns an error. If you are in a subshell and use **setenv** with no arguments, only the variables that have been exported in the subshell are displayed. To list all exported variables that were passed to a subshell use the **printenv** command described in Module 107.

Automatically Exported Variables The commonly used **csh** (local) variables *path*, *term*, and *user* are automatically exported to the environment variables *PATH*, *TERM*, and *USER*. This allows you to perform a **set variable=***value* or a **setenv VARIABLE** *value*.

RELATED COMMANDS Refer to the **set** command described in Module 118 and the **printenv** command in Module 107.

APPLICATIONS

The **setenv** command is used to allow information to be passed from a parent process to a child process. For example, the **vi** editor uses the environmental SHELL and TERM variables to know what type of terminal it is working with and which shell to invoke when you request one.

TYPICAL OPERATION

In this activity you use the **setenv** command to place the COLUMNS variable in the environment. You then start a subshell and view the exported commands using **setenv**. Begin at the shell prompt. The **setenv** command places these and other variables in the environment that is passed to other processes.

1. Type **printenv | col** and press **Return**. Notice all of your environment variables are displayed.
2. Type **setenv COLUMNS 80** and press **Return**. No response is sent to your terminal.
3. Type **printenv | col** and press **Return**. Notice now the COLUMNS variable is in the environment.

```
cj> printenv
.
.
.
COLUMNS=80
```

4. Turn to Module 145 (SV), Module 107 (BSD) to continue the learning sequence.

Module 120
shift

DESCRIPTION

The internal **shift** command reassigns the positional parameters to the next lower positional number. It allows you to *left shift* your positional parameters. Left shifting the positional parameters causes parameter n to become n-1. For example, if you type echo $2 and then perform a shift, you can display the same value again by typing echo $1. The $2 parameter was shifted left one position to $1. The $0 parameter is not shifted because it is not considered a positional parameter.

COMMAND FORMAT Following is the general format of the **shift** command.

```
shift [ n ]
```

Arguments The following describes the argument that may be passed to the **shift** command.

n The number of positions to shift the parameters.

If no *n* is given then shift assumes the value of 1.

FURTHER DISCUSSION Normally the **shift** command shifts parameters left by one position. But you can specify a number or a variable containing a number to shift multiple positions. The following example shows how the **shift** command affects the positional parameters. Use the **set** command to set positional parameters for your current shell. The first command is equivalent to executing a shell script and passing it the 12 positional parameters.

```
cj> set -- pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9 pp10 pp11 pp12
cj> echo "$#: $*"
12: pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9 pp10 pp11 pp12

cj> echo ${11} ${7} ${3} ${1}
pp11 pp7 pp3 pp1

cj> shift
cj> echo "$#: $*"
11: pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9 pp10 pp11 pp12

cj> shift 8
cj> echo "$#: $*"
3: pp10 pp11 pp12

cj> echo $1 $3
pp10 pp12

cj> shift 3
cj> echo "$#: $*"
0:
```

NOTE

The Korn shell added the feature to reference positional parameters higher than nine. The Bourne shell can only reference up to nine positional parameters; therefore, you were forced to use the **shift** command to access any parameters in positions greater than nine.

DIAGNOSTICS AND BUGS If you attempt to **shift** parameters when there are none to shift, you will receive an error message from the shell resembling:

```
ksh: prog: cannot shift
```

RELATED COMMANDS Refer to the **ksh** command described in Module 71 and Parameters and Variables discussed in Module 99.

APPLICATIONS

The **shift** command provides you a method of processing the positional parameters. If you are writing shell scripts, you may want to reassign your positional parameters to known variable names. By using the **shift** command in combination with a while loop and case statement, you can step through all the positional parameters, reassigning them as needed.

TYPICAL OPERATION

In this activity you use the **shift** command to process positional parameters in a while loop. Begin at the shell prompt.

1. Type the following lines, pressing **Return** at the end of each line. You may want to store this in a file for later reference.

```
cj> set -- -x -y -z 30 afile myfile
cj> while [ $# -gt 0 ]   # when there are no more arguments stop loop
do
 case $1 in
   -x) XFLAG="yes" ;;
   -y) YFLAG="yes" ;;
   -z) ZFLAG=$2    ; shift ;; # shift once here for the -z
    *) FILES="${FILES} "$1 ;;
 esac
 shift  # shift the first positional parameter off the argument list.
done
echo "${XFLAG} ${YFLAG} ${ZFLAG} ${FILES}"
```

C Shell

```
cj> set XFLAG; set YFLAG; set ZFLAG; set FILES
cj> set argv=(-x -y -z 30 afile myfile)
cj> while ($#argv > 0)
switch ($argv[1])
 case -x:
   set XFLAG="yes"
   breaksw
 case -y:
   set YFLAG="yes"
   breaksw
```

```
    case -z:
      set ZFLAG=$argv[2]; shift
      breaksw
    default:
      set FILES="$FILES $argv[1]"
    endsw
    shift
  end
  echo "$XFLAG $YFLAG $ZFLAG $FILES"
```

The output looks like this.

```
yes yes 30 afile myfile
```

If you are using the **ksh** you can edit the code by pressing **Escape** and typing **k**. Repeat typing a **k** until the **while** appears on your command line; press **v** to enter **vi** to edit the file. When you finish editing the file type **ZZ** and the command is executed again.

2. Turn to Module 121 to continue the learning sequence.

Module 121
sleep

DESCRIPTION

The external **sleep** command suspends execution for a specified amount of time. The **sleep** command causes the shell to wait for a certain period of time to elapse before the next instruction or command is processed.

COMMAND FORMAT Following is the general format of the **sleep** command.

```
sleep [ n ]
```

Arguments The following argument may be passed to the **sleep** command.

n Specifies the number of seconds to sleep before processing the next command.

FURTHER DISCUSSION The **sleep** command is often used in shell scripts to delay the execution of a periodic check. You may also want to delay the execution of one command. For instance, let's say you need to go to a meeting but you also need a printout after a job finishes in 15 minutes. You could use the following code to print the output file to the printer in 30 minutes

```
cj> ( sleep 1800; lp -dhplj report1 ) &
```

This causes the subprocess to sleep for 1800 seconds (30 minutes) before executing the **lp** command. The ampersand (&) places the subshell in the background.

Another useful example of the **sleep** command is to notify you of an appointment or other important time-relative need. The following example can be used:

```
cj> ( sleep 3600; echo "\007\nMeeting in large conference room!\n" ) &
```

This subshell will remain in background for about 1 hour. Then it will **echo** an audible BELL and the message to your terminal.

NOTE

The seconds specified are an estimate of time. Since the sleep command is a process, it may or may not get enough CPU time per second to perform its time-keeping correctly. Extremely slow response on a system usually causes **sleep** to overestimate its allocated time.

RELATED COMMANDS Refer to the **wait** command described in Module 152.

APPLICATIONS

The **sleep** command has two types of uses. The first is to delay the execution of a command. If you want to wait a few minutes before you send something to the printer but you need to leave your office, you can use **sleep** to delay the print command.

The second use is to use **sleep** inside a loop. You can have an endless loop that checks or monitors another process. By placing a sleep in the loop you can cause the loop to wait before performing the periodic check. This saves CPU cycles for you and the other users.

TYPICAL OPERATION

In this activity you use the **sleep** command to delay a loop that checks for certain users logging in to the system. You may want to select user names not currently logged into the system to perform this activity, then repeat it with user names that are logged in to the system. Begin at the shell prompt.

1. Type the following code, pressing **Return** at the end of each line. You may wish to place this code in a file and use it as a shell script.

```
cj> until who | egrep "bill|nancy"
do
 echo "Sleeping 60 seconds."
 sleep 60
done
echo "\007\n bill|nancy are on"
```

C Shell

```
loop:
who | egrep "bill|nancy"
if ($status !=0) then
  echo "Sleeping 60 seconds."
  sleep 60
  goto loop
endif
echo "^G^M bill|nancy are on"
```

If you wish to use this as a shell script, place it in a file (named *monitor*) using your favorite UNIX editor (vi). Change the modes to 755 and change the string "bill|nancy" to "$1." To execute the new command type **monitor "bill|nancy"** and press **Return**.

The $1 allows you to specify a valid **egrep** expression for monitor to check for in the output from the **who** command.

2. This code executes until the user *bill* or *nancy* logs in to the system. Therefore, you will need to Break out of the loop by pressing **Delete**.
3. Turn to Module 152 to continue the learning sequence.

Module 122
sort

DESCRIPTION

The external **sort** command sorts ASCII data. The input can be from files or the standard input. The output can be a file or the standard output. If multiple input files are given, the data from each file is merged during the sort.

COMMAND FORMAT Following is the general format of the **sort** command.

```
sort [ - ] [ -cmu ] [ -ooutput ] [ -ykmem ] [ -zrecsz ] [ -dfiMnr ] \
     [ -btx ] [ +pos1 [ -pos2 ] ] [ file_list ]
```

BSD (Berkeley)

```
sort [ - ] [ -cmu ] [ -ooutput ] [ -dfinr ] [ -btx ] \
     [ +pos1 [ -pos2 ] ] [ -Tdirectory ] [ file_list ]
```

Options The following list describes the options and their arguments that may be used to control how **sort** functions.

- Forces **sort** to read from the standard input. Useful for reading from pipes and files simultaneously.

-c Verifies that the input is sorted according to the other options specified on the command line. If the input is sorted correctly then no output is provided. If the input is not sorted then **sort** informs you of the situation. The message resembles this.

```
sort: disorder: This line not in sorted order.
```

-m Merges the sorted input. **sort** assumes the input is already sorted. **sort** normally merges input as it sorts. This option informs **sort** that the input is already sorted, thus **sort** runs much faster.

-o*output* Sends the output to file *output* instead of the standard output. The *output* file may be the same name as one of the input files.

-u Suppress all but one occurrence of matching keys. Normally, the entire line is the key. If field or character keys are specified, then the suppressing is done based on the keys.

-y*kmem* Use *kmem* kilobytes of main memory to initially start the sorting. If more memory is needed, **sort** automatically requests it from the operating system. The amount of memory allocated for the sort impacts the speed of the sort significantly. If no *kmem* is specified, **sort** starts with the default amount of memory (usually 32K). The maximum (usually 1 Megabyte) amount of memory may be allocated if needed. If 0 is specified for *kmem*, the minimum (usually 16K) amount of memory is allocated.

-z*recsz* Specifies the record size used to store each line. Normally the *recsz* is set to the longest line read during the sort phase. If the -c or -m options are specified, the sort phase is not performed and thus the record size defaults to a system size. If this default size is not large enough, **sort** may abort during the merge phase. To alleviate this problem you can specify a *recsz* that will allow the merge phase to run without aborting.

Ordering Rule Options

-d Specifies dictionary sort. Only blanks, digits, and alphabetic characters are significant in the comparison.

-f Fold lowercase letters to uppercase. Ignores the significance of upper and lowercase ASCII characters.

-i Ignore characters outside the ASCII range of 040 (octal) and 0176 (octal). Only alphabetic characters, blanks, digits, and punctuation are used for comparison (printable characters). Control characters are ignored. This is only valid for nonnumeric sorts.

-M Compare fields as months. The first three nonblank characters are folded (see -i option) to uppercase and compared. Thus January is compared as JAN. JAN precedes FEB, and fields not containing months precede JAN. The -b option is placed in effect automatically.

-n Sorts the input numerically. The comparison is based on numerical value instead of alphabetic value. The number field used for comparison can contain optional blanks, optional minus signs, optional decimal point, and zero or more digits. The -b option is placed in effect automatically. Exponential numbers are not sorted correctly.

-r Reverse the order of the output.

Restricted Sort Key Options

+*pos1* Specifies the beginning position of the input line used for field comparison. If *pos1* is not specified then comparison begins at the beginning of the line. The *pos1* position has the notation of *f.c*. The *f* specifies the number of fields to skip. The *c* specifies the number of characters to skip. For example, 3.2 is interpreted as skip three fields and two characters before performing comparisons. Omitting the *.c* portion is equivalent to specifying .0. Field one is referred to as position 0. If *f* is set to 0 then character positions are used for comparison.

-*pos2* Specifies the ending position of the input line used for field comparison. If *pos2* is not specified then comparison is done through the end of the line. The *pos2* position has the notation of *f.c*. The *f* specifies to compare through field *f*. The *c* specifies the number of characters to compare through after field *f*. For

example, -4.3 is interpreted as compare through three characters after the end of field four. Omitting the *.c* portion is equivalent to specifying .0.

-b Ignores leading blanks when using the restricted sort key positions (*+pos1* and *-pos2*). If the -b option is placed before the first *+pos1* argument, then it applies to all *+pos1* arguments. The -b option can be attached to each *pos* string to affect only that field.

-t*c* Use character *c* as the field separator. Multiple *c*'s may be specified to represent empty fields surrounded by separators.

Both these options and any of the ordering rule options (d, f, i, M, n, and r) can be attached to position arguments. For example,

```
cj> sort -t: +2.0n -3.0 +0.0bdf -1.0 /etc/passwd
```

This command sorts the passwd file with the primary key field being field 3. The secondary key is based on field 1. Field 3 is sorted as numbers, while field 1 is sorted in dictionary order, ignoring uppercase versus lowercase characters.

BSD (Berkeley)

-T*directory* Specifies the directory to store temporary files.

Arguments The following arguments may be passed to the *sort* command.

file_list One or more input files to be read and sorted.

If no files are specified the standard input is read.

FURTHER DISCUSSION The **sort** command can read from the standard input and write to the standard output. Thus it can be used in pipes or have its output redirected as desired. For example, sort can be used in the following combinations of commands.

```
sort infile > outfile
sort infile | lp
who | sort
sort infile -o infile
```

In the first command **sort** reads and sorts *infile* and displays the result on standard output, which has been redirected to *outfile*. In the second command **sort** reads and sorts *infile* and displays the results on standard output, which is piped to the **lp** command. The third command pipes the output of the **who** command to the input of the **sort** command. The *sort* then displays its output on the standard output (your screen). In the last command **sort** reads from the file **infile** and writes the sorted output to the file **infile**.

By default the **sort** command sorts based on entire lines. It compares the first character of two lines. If the first character is the same, then the second character is compared, and so on until the entire line is compared or the correct order is determined. Leading blanks (spaces and tabs) are considered valid characters to compare. Thus a line beginning with a space precedes a line beginning with the letter A. This is because the space is octal 040 in the ASCII sequence and an A is octal 101.

If the ordering rule options precede the sort key options, they are globally applied to all sort keys. For example,

```
sort -r +2 -3 infile
```

reverses the order based on field 3. If the ordering rule options are attached to a sort key option, they override the global ordering options for only that sort key. For example,

```
sort -r +2d -3d infile
```

sorts field 3 by dictionary comparison but sorts the rest of the line using reverse comparison.

Multiple sort keys may be used on the same command line. The first key is used to sort the lines. If two lines have the same value in the same field, *sort* uses the next set of sort keys to resolve the equal comparison. For example,

```
sort +4 -5 +1 -2 infile
```

means to sort based on field 5 (+4 –5). If two lines have the same value in field 5, sort those two lines based on field 2 (+1 –2). If all the sort keys match, then the rest of the line is used for comparison.

DIAGNOSTICS AND BUGS If equal keys are referenced **sort** does not necessarily use the relative order of the lines.

RELATED COMMANDS Refer to the **comm, join**, and **uniq** commands described in modules 22, 69, and 149.

RELATED FILES The **sort** command uses a temporary file /usr/tmp/stm???.

APPLICATIONS

You use the **sort** command to sort data alphabetically or numerically, in ascending or descending order. You can sort based on entire lines, fields, or character columns. You can merge files using **sort** and remove duplicate lines with it.

TYPICAL OPERATION

In this activity you use the **sort** command to sort the /etc/passwd file in using various sort keys and ordering rules. Begin at the shell prompt.

1. Type **sort -t: +2 /etc/passwd** and press **Return**. Notice the order of the output is based on the third field of the passwd file, user IDs. But notice that the order of the numbers are not correct.
2. Type **sort -rn -t: +2 /etc/passwd** and press **Return**. Notice the order of the numbers is now correct, but in reverse order. The -n option informs **sort** to compare the specified field as numbers, not ASCII characters. The r option reverses the order of the sort.
3. Type **sort -t: +5 -6 +0 -1 /etc/passwd** and press **Return**. The output is now sorted by field 6, then by field 1 if necessary.
4. Turn to Module 149 to continue the learning sequence.

Module 123
spell

DESCRIPTION

The external **spell** command checks input for spelling errors. It compares the words it reads from input to words in its spelling list. If a word is not located in its list, then the word is printed on the standard output. **spell** adds suitable prefixes, suffixes, and inflections to derive the correct spelling of a word.

Most **troff**, **nroff**, **tbl**, **eqn**, and **pic** constructions are ignored by **spell**. The **deroff** command is executed by **spell** to remove these constructs.

By default **spell** follows chains of **troff** and **nroff** include files. Include files are preceded by .so or .nx constructs. If the filename begins with /usr/lib, **spell** ignores the file and does not read it. Most /usr/lib files are **mm** macros created by vendors, not by users. If the -l option is specified, then all include files are read, even the /usr/lib files.

The spelling list file was created from many sources. Although it is not a Webster's Dictionary, it is very effective with respect to proper names and technical words. Unfortunately, the coverage of special professions such as biology, chemistry, and medicine is lacking to say the least.

The **hashmake** command converts a list of words to a list of nine-digit hash codes. The list of words are read from the standard input. The list of hash codes are written to the standard output. The output of **hashmake** is an intermediate file used by **spellin** to create the final spelling list (*hlist*).

The **spellin** command converts hash codes to a hashed compressed spelling list. The hash codes are read from the standard input. The compressed spelling list is written to the standard output. The output of **spellin** is the final spelling list used by **spell**; the file is referred to as the *hlist*.

The **hashcheck** command converts a compressed spelling list to a list of hash codes. The compressed spelling list is read from the standard input. The hash code list is written to the standard output. You use the **hashcheck** command to verify that the compressed spelling list is valid.

BSD (Berkeley)

The **spellin** command creates a hashed compressed spelling list. The ASCII word list is read from the standard input. If a filename is given it is assumed to be an existing spelling list file (*hlist*). The output, a new or updated spelling list, is written to the standard output.

The **spellout** command searches the given hashed spelling list (*hlist*) for each word read from the standard input. If the list does not contain one of the input words, the word is written to the standard output. With the -d option specified, the word is written to the standard output if it does exist in the hashed list.

COMMAND FORMAT Following is the general format of the **spell** command.

```
spell [ -blvx ] [ +local_file ] [ file_list ]
/usr/lib/spell/hashmake
/usr/lib/spell/spellin n
/usr/lib/spell/hashcheck word_list
```

BSD (Berkeley)

```
spell [ -bvx ] [ -d hlist ] [ -h spellhist ] [ -s hstop ] [ file ]
spellin [ list ]
spellout [ -d ] list
```

Most users only use the **spell** command. The other commands are usually used by the system administrator to add words to the spelling list.

Options The following list describes the options and their arguments that may be used to control how **spell** functions.

-b Check for British spelling. The *ise* ending is required in words such as *standardize*. Such spelling as *colour* and *centre* are checked.

-l Causes the files referenced in a troff document to be read even if their pathname begins with /usr/lib. Normally **spell** will check the spelling of chained files in a troff document, unless the pathname of the file begins with /usr/lib. Chained files are .so and .nx requests in a troff document.

-v Displays all words not literally in the spelling list. All reasonable alternative words are displayed if any exist.

-x Displays possible stems, prefixed with an equal sign (=), for each word being checked.

+local_file Words listed in *local_file* are removed from spell's output of misspelled words. The *local_file* contains a sorted list of words, one per line. You can use this file to store a list of correctly spelled words unknown to the normal spelling list (*hlist*).

BSD (Berkeley)

-d *hlist* Use *hlist* as the spelling list file instead of the default. The default is /usr/dict/hlist[ab].

-h *spellhist* Use *spellhist* as the history file instead of the default. The default is /dev/null. The history file contains a copy of all words spelled incorrectly according to spell.

-s *hstop* Use *hstop* as the stop list file instead of the default. The default is /usr/dict/hstop.

See the following section "Control Files" for definitions of these three files.

Arguments The following list describes the arguments that may be passed to the **spell** command.

file_list One or more files read and each word checked for correct spelling.

If no files are listed the standard input is read for words.

Control Files The following files are used by **spell** to verify the correct spelling of the input word list.

hlist The list of correctly spelled words in hashed format.

spellhist Copies of all output. This provides the system administrator with a list of words that have been rejected by **spell** as incorrect. Technical words may be taken from the list and added to the list of correctly spelled words (*hlist*).

hstop The list of words used to filter out words that pass the normal spell checking. For example, their=they-ey+eir.

The following environment variables may be set to override the default files used by **spell**. They correspond to the BSD -d, -h, and -s options.

D_SPELL=*hlist* Defines your *hlist* file instead of the default. The default is /usr/lib/spell/hlist[ab].

H_SPELL=*spellhist* Defines your *spellhist* file instead of the default. The default is /usr/lib/spell/spellhist.

S_SPELL=*hstop* Defines your *hstop* file instead of the default. The default is /usr/lib/spell/hstop.

Generating New *hlist* **To generate a new** *hlist* (correctly spelled words) perform the following steps.

1. Create a list of correctly spelled words. Use **vi** or another favorite editor. Name the file *goodwords*.
2. Type **cat goodwords | /usr/lib/spell/hashmake | sort -u > gw1** and press **Return**. This creates a temporary hash list of the new words.
3. Type **cat gw1 | /usr/lib/spell/spellin 'cat gw1 | wc -l' > hlist** and press **Return**. This creates a compressed list of correctly spelled words (a spelling list).

To check the hash list.

4. Type **cat hlist | /usr/lib/spell/hashcheck > gw2** and press **Return**. This creates a list of hash codes for the corresponding compressed spelling list.
5. Type **diff gw1 gw2** and press **Return**. This verifies that the new spelling list (hlist) is correct. If the hash list (*hlist*) is correct no output is displayed.
6. Type **rm gw1 gw2** and press **Return** to remove the work files.

You now have a correct and functional *hlist* file to use for your own words.

BSD (Berkeley)

1. Create a list of correctly spelled words. Use **vi** or another favorite editor. Name the file *goodwords*.

2. Type **cat goodwords | spellin > hlist** and press **Return**. This creates a hashed spelling list.

To check the hash list.

3. Type **cat goodwords | spellout hlist** and press **Return**. Any words in *goodwords* not in the hash list (*hlist*) are displayed on your terminal.

DIAGNOSTICS AND BUGS New installations will probably find it necessary to monitor the output of **spell** on local documents to collect a local spelling list. Once the list is collected, a new local list can be added to **spell**'s list of words.

RELATED FILES The **spell** command reads from files or from the standard input. It writes any misspelled words to the standard output. It uses different files for spell checking located in the /usr/lib/spell directory.

APPLICATIONS

The **spell** command is used to check your documents for spelling errors. Unfortunately, it is not interactive in correcting the misspelled words, nor does it suggest correct spellings.

TYPICAL OPERATION

In this activity you use the **spell** command to check the spelling of words in a file. Begin at the shell prompt.

1. Type **spell file1** and press **Return**. If no output is returned to your screen, then no words are misspelled in the *file1* file. The output should resemble the following.

```
cj> spell file1
file1
```

2. If you would like to see more misspelled words type **spell /etc/passwd** and press **Return**.
3. Turn to Module 140 to continue the learning sequence.

Module 124
split

DESCRIPTION

The external **split** command splits a file into smaller files based on a specified number of lines. Each of these smaller files are equal in size, with the exception of the last one created. It is the remainder of the original. Your original file is not changed by **split**.

The **split** command reads input from a file or the standard input and creates multiple output files. Each file contains *n* lines of the original. If you do not provide *n*, **split** uses the value of 1000.

If you provide the *newname* argument, the destination files are named *newname*XX. Where XX is aa for the first file, ab for the second, and continues until the file zz. That's a total of 676 files you can generate if you divide your input into small enough sizes. When using *newname*, you must use a name two characters shorter than the maximum allowed for filenames. Maximum filename length is 100; therefore, you can only use filenames of 98 characters for *newname*. If you do not provide a *newname* argument, the destination files are named xXX. **split** uses the x as a *newname*.

COMMAND FORMAT The general format of the **split** command follows.

```
split [ -n ] [ - ] [ file [ new_name ]
```

Options The following option may be used to control how **split** functions.

-n Causes the output files to contain *n* lines.

If *n* is not specified, **split** uses 1000. The input is split into files containing 1000 line each.

Arguments The following list describes the arguments that may be passed to the **split** command.

- Causes **split** to read from the standard input.

file The name of the file **split** reads and divides into *n* or 1000 line files.

If no file is given on the command line, **split** will read from the standard input.

newname The base part of the name used for all output files. An extension is added to *newname* for each file created. The extension is made up of two alpha characters. The first file extension is "aa," then "ab," and so on until the original input is completely divided.

If *newname* is not specified, the output is written to a file with a base part of "x" and the normal extensions. Thus the default output filenames are xaa, xab, and so on.

RELATED COMMANDS Refer to the **csplit** command described in Module 28.

RELATED FILES **split** places its output in files with an extension of two characters. The characters begin with "aa," the next file is "ab," and so on until the entire input has been split and stored in multiple files.

APPLICATIONS

You may find the **split** command helpful in dividing large data files into smaller, more manageable files. Since the extensions added by **split** are a sortable sequence, you can process the new files using shell looping structures.

There are some commands that cannot handle extremely large files; therefore, you may have to **split** the input for these commands into more manageable blocks.

TYPICAL OPERATION

In this activity you use the **split** command to divide the standard input into separate output files. Begin at the shell prompt.

1. Type **ls /bin | split -20 - bin** and press **Return**. Notice no output is returned from the **split** command.
2. Type **ls -x bin*** and press **Return** to display the new files created by the **split** command. Each of these *binaX* files contain a portion of the output from the **ls /bin** command.
3. Type **cat binaa** and press **Return** to display the contents of the first output file created by **split**.
4. Type **rm bina*** and press **Return** to remove all of the files beginning with bin.
5. Turn to Module 28 (SV), Module 69 (BSD) to continue the learning sequence.

Module 125
stop (csh, ksh)

DESCRIPTION

The internal **stop** command stops the execution of a background job. If you have a job running in background and type **stop**, the current background job is stopped (placed in a suspended state). You can then kill the job, bring it into foreground, or place it in background again.

Korn Shell

The **ksh** does not implement **stop** as an internal command. You have to create an alias to perform the function. Simply place the following line in your .profile.

```
alias stop="kill -STOP"
```

COMMAND FORMAT Following is the general format of the **stop** command.

```
stop [ %job ... ]
kill -STOP [ %job ... ]
```

The **ksh** only supports the second format. But you can set up an alias named stop to perform the kill command.

Arguments The following arguments may be passed to the **stop** command.

%job A job number associated with a process. If the interactive option is turned on, *job* may be one of the following:

%num	A job number associated with a process.
%string	The job whose name begins with *string*.
%?string	The job whose name contains *string*.
%+	
%%	Current process, the last process you suspended.
%-	Previous process, the process suspended before the current job.

The current background job is affected if you do not specify a job number. The current background job is the last job placed in background.

RELATED COMMANDS Refer to the **bg**, **fg**, **jobs**, **kill**, and **suspend** commands described in modules 9, 51, 68, 70, and 129.

APPLICATIONS

The **stop** command can be used to stop the execution of a job you have placed in background, thus allowing you to control whether a job should continue to run. Once you have stopped a job you can return it to background for execution, bring it into the foreground for interactive execution, or terminate it with the **kill** command.

TYPICAL OPERATION

In this activity you use the **stop** command to stop a job you have placed in background mode. Begin at the shell prompt.

1. Type **ls -R / > /dev/null &** and press **Return**. This places a recursive **ls** in background.
2. Type **stop %1** and press **Return**. The **ls** command you just started is stopped.

```
cj> stop %1
[1] + Stopped                    ls -R > /dev/null
```

3. Type **jobs** and press **Return** to display the stopped job.
4. Type **kill %1** and press **Return** to kill the **ls** command.

```
cj> kill %1
[1]   Terminated                 ls -R > /dev/null
```

5. Turn to Module 70 to continue the learning sequence.

Module 126
stty

DESCRIPTION

The external **stty** command sets terminal I/O options for the standard input device. Your standard input device is usually a /dev/ptyXX device driver. If you are on a network, it might be a /dev/ttyXX device driver. Regardless of how you are connected to the UNIX operating system, you have a terminal I/O device driver. The **stty** command can be used to set and reset how the "tty" device driver functions.

The most common use of the **stty** command is to reset your *erase* and *kill* characters. Some people reset a few other options, but for the most part, the default settings handle all of the typical user's requirements.

COMMAND FORMAT Following is the general format of the **stty** command.

```
stty [ -ag ] terminal-options
```

BSD (Berkeley)

```
stty terminal_options
```

Options The following options may be used to control how **stty** functions.

-a	Displays all terminal option settings.
-g	Displays the current terminal option settings in a format you can use as an argument to another **stty** command.

Option Arguments The following are special arguments that perform option type functions on BSD.

BSD (Berkeley)

all	Displays all normally used options.
everything	Displays every setting **stty** knows about.
speed	Displays the terminal/line speed (baud rate).
size	Displays the number of rows and columns on the terminal.

Arguments The following list describes the arguments that may be passed to the **stty** command. These arguments are referred to as *terminal-options*. The terminal options enclosed in parentheses are the opposite settings. For example, ocrnl maps Return to new-line and (-ocrnl) unmaps Return to new-line. Not all terminal options have two settings.

NOTE

Only the more common terminal-options are discussed. For a complete list refer to your vendor's User's Reference Manual.

Control Modes These terminal options control how your terminal and the computer communicate. Most of these options are set automatically when you log in to the system. Resetting these options may cause your terminal to perform strangely and/or even hang or disconnect.

0	Causes the device driver to hang up (disconnect) the phone immediately.
baud	Sets the baud rate of the device driver to *baud*. Supported baud rates vary depending on the hardware terminal interface boards. The baud rate must match the baud rate of your terminal. Popular baud rates are: 110, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400.
cs*n*	Sets the character size to be *n* bits. It takes 7 bits to create one ASCII character. This size is used for transmitting the characters between your terminal and the system. The parity bit is not included in the character size.
hup (-hup) hupcl (-hupcl)	Causes the terminal device driver to hang up (do not hang up) the phone when the last process attached to the device terminates. For example, when you log off the system your shell terminates and the driver hangs up the phone line automatically.

BSD (Berkeley)

0	Same as SV
baud	Same as SV
new	Enable new tty device driver.
nohang (-nohang)	Do not (do) send hang-up signal if carrier drops.
pass8	All 8 bits are passed through on input.
(-pass8)	Strips the 0200 bit on input except in raw mode.
tandem (-tandem)	Enable (disable) flow control.
tostop	Background jobs stop if they attempt to write to the terminal (standard output). Like loblk on SV.
(-tostop)	Output from background jobs is allowed to the terminal.

Input Modes These settings are used to control how the device driver responds to certain character sequences received from your terminal keyboard. If you press a key on your terminal or if the line has noise on it, these options can be set to detect or ignore the signals.

brkint (-brkint)	The device driver sends (does not send) an interrupt signal to your current program on the UNIX system when you press the break key on your terminal.
icrnl (-icrnl)	Converts (does not convert) the ASCII CR (carriage return) character to the ASCII NL (new-line) character on input.

ignbrk (-ignbrk)	Ignores (does not ignore) the break signal on input. The break signal is produced on your terminal by pressing the break key.
igncr (-igncr)	Ignores (does not ignore) the ASCII CR (carriage return) character on input.
ignpar (-ignpar)	Ignores (does not ignore) parity errors. If parity is enabled then each character is checked for the correct parity.
inlcr (-inlcr)	Converts (does not convert) the ASCII NL (new-line) character to the ASCII CR (carriage return) character on input.
inpck (-inpck)	Enables (disables) parity checking on input bytes.
istrip (-istrip)	Strips (does not strip) the high order bit from the input character. The input bit is thus reduced to seven bits. The normal ASCII character set is made up of only seven bits. The eighth bit is needed for extended ASCII or hardware that can utilize the eighth bit.
iuclc (-iuclc)	Converts (does not convert) uppercase characters to lowercase characters on input.
ixany (-ixany)	Enables (disables) any input character to cause output that has been suspended with the stop character (Ctrl-S) to resume.
ixoff (-ixoff)	Enables (disables) input start and stop flow control. The system and the modem use the start and stop characters to control character transmissions. When the input queue is almost empty, the system sends a start character. When the input queue is almost full, the system sends a stop character.
ixon (-ixon)	Enables (disables) the keyboard to control output flow control. This allows you to start and stop output to your screen using the Ctrl-S (stop) and Ctrl-Q (start) characters. The Ctrl-S stops output and the Ctrl-Q resumes output.

BSD (Berkeley)

None of the above settings are supported in BSD. BSD implements the following settings.

decctlq (-decctlq)	After output is suspended, usually by Ctrl-S, only the start character (any character), usually Ctrl-Q, will start the output.
pendin (-pendin)	Input is (not) pending. A switch has occurred from cbreak to cooked and input is waiting. It is re-input by the driver when a new read becomes available from the application program.

Output Modes These terminal options control the sending of data from the computer to your terminal screen.

ocrnl (-ocrnl)	Converts (does not convert) an ASCII CR (carriage return) to an ASCII NL (new-line) on output.
olcuc (-olcuc)	Converts (does not convert) lowercase characters to uppercase characters on output.

onlcr (-onlcr) Converts (does not convert) an ASCII NL (new-line) to an ASCII CR (carriage return) on output.

onlret (-onlret) Maps (does not map) the ASCII NL (new-line) character to be the CR-NL (carriage return, new-line) character sequence on output. The carriage return causes the cursor to return to the beginning of the line. The new-line forces the cursor to the next line. If the - (hyphen) is used then the cursor moves to the next line but does not return to the beginning of the line.

onocr (-onocr) Does not (does) transmit the ASCII CR (carriage return) character when it is the first character on the output line.

opost (-opost) Post-processes (does not post-process) the output sent from the computer. The post-processing refers to the conversions and mappings performed by the other Output Modes discussed in this section. If post-processing is not performed, all output is transmitted unchanged.

BSD (Berkeley)

None of the standard output modes are supported by BSD.

flusho (-flusho) The output is (not) being discarded. Usually caused by the user pressing Ctrl-O.

litout (-litout) Do not (do) perform output processing. Normal output processing involves inserting delays, timing sequences, and other control sequences.

Local Modes These terminal options control how the device driver processes the input from the terminal before sending it to your program running on the computer.

echo (-echo) Echoes (does not echo) back each character you type on your keyboard. Normally, on a UNIX operating system, the characters you type are sent to the computer and are sent back to your terminal so you can see what you typed. If your terminal performs local echoing of input, then you can set this to -echo and set your terminal to echo mode.

echoe (-echoe) Echoes (does not echo) the erase (see Control Characters section) character as the backspace-space-backspace character sequence. The previous character is removed from your terminal screen. The icanon option (see below) must be set. If the echo option is not set, then the erase character is sent as space-backspace.

echok (-echok) Echoes (does not echo) the ASCII NL (new-line) character after you press the kill (see Control Characters section) character.

echonl (-echonl) Echoes (does not echo) the ASCII NL (new-line) character. Ignores the setting of the echo option. The icanon option must be set.

isig (-isig) Enables (disables) the checking of input characters for signals. Signals are the control characters (see Control Characters section) that have special meaning.

noflsh (-noflsh) Disables (enables) the flushing of input when an interrupt, quit, or switch signal is received from the terminal. Any text typed from a terminal that

	had not been sent to a running program is discarded if one of the defined keys is pressed.
xcase (-xcase)	Processes (does not process) uppercase and lowercase characters. If processing is enabled, then lowercase characters are entered at the keyboard by pressing the desired key. Uppercase characters are entered by preceding the character with a \ (backslash).

BSD (Berkeley)

echo (-echo)	Same as SV
cbreak (-cbreak)	Passes each character on to application program as received. Disable (enable) erase or kill processing is performed.
crtbs	Echo backspaces on erase characters.
crterase (-crterase)	Same as SV echoe
crtkill (-crtkill)	Does not (does) echo the ASCII NL (new-line) character after you press the kill (see Control Characters section) character. The current command line is erased using crterase.
ctlecho (-ctlecho)	Echo (do not echo) control characters as "^X" where X is the character. Two backspaces are echoed after an EOT (Ctrl-D) is pressed.
prterase (-prterase)	(Do not) Print erased characters within "\" and "/".

Control Characters Certain control characters are interpreted by the tty device driver as special characters to perform specific functions. The **stty** command allows you to change the definition of these options so you can use a control key you desire to perform the related function. The form of these terminal options is a little different from the other forms. The general form is

```
Terminal_Option  Ctrl-Character
```

The terminal_option is one of the following options and the ctrl-character is a character you wish to use to perform the function related to the option. These characters are control characters. To enter a control character, you type the control character directly on the command line, although some control characters will be interpreted by the shell and the device driver. Another way to enter control characters is to place a caret (^) before the letter you want to be a control character. For example,

```
erase ^h
```

sets the erase character to Ctrl-H. The ^ is a caret not a Ctrl-H.

The value of *c* may be set to NULL (undefined) by using the "^'" notation. The delete character is represented as "^?". Escape is represented as "^[".

ctab	Defines ^*c* as the control tab character for applications using synchronous lines (line mode is enabled).
eof ^*c*	Defines ^*c* as the EOF (end-of-file) character. If the specified character is received as the first character on a line, then the end-of-file signal is sent

	to the program on the UNIX system. If the character is received in-line, all previous text entered on the current line is sent to the program. Default is Ctrl-D.
eol ^*c*	Defines ^*c* as the EOL (end-of-line) character. The eol is an additional end-of-line character. The normal EOL is the ASCII LF or NL (linefeed or new-line). The eol character is used to delimit a line of input text. Default is NULL.
erase ^*c*	Sets the erase character to ^*c*. When you press the ^*c* character on your terminal, the device driver backspaces one character. The default value of ^*c* is normally #, which is often replaced by Backspace (Ctrl-H).
intr ^*c*	Character ^*c* is set to be the interrupt signal. When you press ^*c* the interrupt signal is sent to each process currently controlled by your tty device. Background processes are not affected. The normal key defined is Delete (Del, 177 octal). If the brkint terminal option is set, the Break key will also generate the interrupt signal.
kill ^*c*	Sets the kill character to ^*c*. When you press ^*c* the entire line of input is deleted. Your cursor is moved to the next line or beginning of same line if echok is disabled. The default kill character is @, which is often reset to Ctrl-U.
min *n*	The minimum number of characters required to satisfy a read if the -icanon terminal option is set. The *n* is a digit specifying the number of characters to be read from the terminal.
quit ^*c*	Sets character ^*c* to generate the quit signal. This signal is much like the interrupt signal except abnormal termination results in any currently running programs. The default quit character is Ctrl-\.
time *n*	The time delay value used in the -canon terminal option. The default value of *n* is 0.10 second.

BSD (Berkeley)

The value of *c* may be set to NULL (undefined) by using "u" or "undef."

eof erase intr kill quit	Same as SV
brk *c*	Set the break character to *c*. The default is NULL (undefined).
cols *n* columns *n*	Sets the number of columns assumed on the terminal to *n*.
rows *n*	Sets the number of rows assumed on the terminal to *n*.
start *c*	Set the start character to *c*. The default is Ctrl-Q. This allows you to start output to your terminal that has been stopped by the stop character (Ctrl-S).

stop *c*	Set the stop character to *c*. The default is Ctrl-S. This allows you to stop output to your terminal.
tilde (-tilde)	(Do not) Convert the tilde character (~) to acute accent (‘).

Combination Modes The following terminal options are aliases for common combinations of previously defined terminal options.

nl	Unsets the incrnl and onlcr terminal options.
-nl	Sets incrnl and onlcr and unsets inlcr, igncr, ocrnl, and onlret.
LCASE (-LCASE) lcase (-lcase)	Sets (unsets) the xcase, iuclc, and olcuc terminal options.
tabs (-tabs or tab8)	Expands tabs to the equivalent number of spaces when on output.
ek	Resets the kill character to @ and the erase character to #.
sane	Resets the tty device driver terminal options to some reasonable values. This is useful if you have to kill a program with a kill -9 PID command and the program has reset some of the terminal options.

BSD (Berkeley)

ek	
lcase (-lcase)	
nl (-nl)	
raw (-raw or cooked)	Same as SV
crt	Sets options for a CRT. The crtbs, crtecho (if the baud rate >=1200), crterase, and crtkill are set.
dec	Sets all modes suitable for Digital Equip. Corp. users (erase=Ctrl-?, kill=Ctrl-U, intr=Ctrl-C, and the decctlq and newcrt modes are set).

NEW TTY DRIVER MODES The following modes (tty settings) are only applicable to the new BSD tty device driver. Usually they are not changed. The new device driver is used by specifying stty new.

BSD (Berkeley)

susp *c*	Set the suspend process character to *c*. The default is Ctrl-Z.
dsup *c*	Set the delayed suspend process character to *c*. The default is Ctrl-Y.
rprnt *c*	Set the reprint line character to *c*. The default is Ctrl-R. Redraws the command line. Useful when output from a background job or other user has garbled your command line.
flush *c*	Set the flush output character to *c*. The default is Ctrl-O. Flushes all currently held output to your terminal.

werase *c*	Set the erase word character to *c*. The default is Ctrl-W. If you press the Ctrl-W on the command line, the previous word is erased.
inext *c*	Set the literal next character to *c*. The default is Ctrl-V. This allows you to enter a character with special meaning to the device driver on the command line. For example, Ctrl-R normally redraws the command line. But Ctrl-V Ctrl-R inserts the Ctrl-R on the command line.

RELATED COMMANDS Refer to the **tabs** command described in Module 130.

RELATED FILES The **stty** command sends control signals to the device driver you are currently logged into the system through.

APPLICATIONS

The **stty** command has many options you can use to change how the computer system communicates with your terminal. Most of the terminal options described are set automatically to the proper setting. You can, however, reset some of the options to customize how the system communicates with your terminal. For example, the local modes, control characters, and the combination modes are often used to reset how the terminal device driver functions, thus changing how the terminal acts.

The most common use of the **stty** command is to reset your *erase* and *kill* characters. Some people reset a few other options but, for the most part, the default settings handle all of the typical user's requirements.

The *echo* option is sometimes used in shell scripts to turn off echoing when you want to read in a password or some type of encryption key and you do not want the input to be displayed on the screen. For example,

```
cj> cat readpasswd
stty -echo
echo "Enter database password: \c"
read PSWD
stty echo
```

might be used to read in a password without having the password displayed while it is being typed by the user.

The **stty** command may also be used by the super-user to change the settings of a tty other than the one currently logged in on. For example,

```
cj> stty 9600 < /dev/tty09
```

resets the baud rate for tty09 to 9600. The command

```
cj> stty -a < /dev/tty09
```

displays all settings for tty09. On some systems you have to use output redirection (>) instead of input redirection (<) to access a tty other than yours.

TYPICAL OPERATION

In this activity you use the **stty** command to display the current settings of your tty device driver. Then change options to show how the **stty** command can affect how your terminal communicates to the UNIX system. Begin at the shell prompt.

1. Type **stty** and press **Return**. The output describes the current settings of how the terminal device driver for your terminal is set.

```
cj> stty
speed 9600 baud; -parity hupcl susp = ^Z; dsup = ^Y
erase = #; kill = @; swtch = ^@;
icrnl onlcr tab3
echo -echoe echok
```

2. Now type **stty -echo** and press **Return**.

3. Type **stty echo** and press **Return**. Notice the characters you type are not echoed on your terminal screen. This is because the *echo* terminal option has been disabled. Thus the system is not echoing your characters back to your terminal. The **stty echo** command enables the echo feature.

4. Type **stty erase ^h kill ^u** and press **Return**. Note the ^ is a caret not a control character representation.

5. Type **stty** and press **Return** to display your terminal settings again. Depending on the original settings displayed in step one you should see about the same definitions except for the kill character.

```
cj> stty
speed 9600 baud; -parity hupcl susp = ^Z; dsup = ^Y
erase = ^H; kill = ^U; swtch = ^@;
icrnl onlcr tab3
echo -echoe echok
```

You have completed the learning sequence.

Module 127
su

DESCRIPTION

The external **su** command lets you become another user without logging off the system. The password of the new user name must be supplied unless you are super-user. Super-user can su to any user name without knowing the user's password. You can use **su** to log in as a new user with a new user ID and environment or to just take on a new user ID.

COMMAND FORMAT Following is the general format of the **su** command.

```
su [ - ] [ user_name [ -c cmdstring | -r cmdstring ]
```

BSD (Berkeley)

```
su [ -f ] [ - ] [ user_name ]
```

Options The following list describes the options that may be used to control how **su** functions.

-	Informs su to use the new user_name's environment as if you had logged in as that user. The default is to pass your current environment with the possible exception of $PATH. The - specifies that arg0 should have a hyphen placed in front of it. This tells the login shell that you are initially logging in to the system. The login shell will check arg0 for a hyphen and, if found, execute /etc/profile and the new user_name's .profile.

C Shell

The C shell reads the .login and .cshrc shell when the shell is a login session.

-c	Su passes the string that follows the -c as a command line to the new user_name's shell. Refer to *cmdstring* in the argument section.
-r	Same as -c except a restricted shell is executed.

BSD (Berkeley)

-f	Causes the **csh** to skip the execution of the .cshrc file. This allows for a faster login but does not set up all environment variables residing in the .cshrc file.
-	Forces a simulated full login.

Arguments The following list describes the arguments that may be passed to the **su** command.

user_name A valid user name (login name).

cmdstring A shell command that will be passed to the new environment and executed by the new user_name shell.

BSD (Berkeley)

If no *user_name* is specified, "root" is assumed. A number sign is substituted in place of the normal super-user prompt.

FURTHER DISCUSSION **su** executes a new shell or program defined in the shell field (field 7) of the /etc/passwd file of the specified user. The real and effective user ID and group ID will be set according to the new user. To exit the new environment press Ctrl-D or type **exit**. You will return to your previous shell.

Each attempt to **su** to a different *user_name* is logged in the su log file, /usr/adm/sulog.

BSD (Berkeley)

Without any options **su** starts a new shell as specified in the shell field of the /etc/passwd file without changing your current directory. Your environment does not change except for HOME and SHELL, which are reset to the values found in the /etc/passwd file.

RELATED COMMANDS Refer to **env**, **login**, **passwd**, and **ksh** described in modules 41, 77, 99, and 72, respectively.

RELATED FILES The following files are read or updated by the **su** command.

/etc/motd	Message of the day file, displayed during login.
/etc/passwd	System password file.
/etc/profile	System profile, read by login shell to set up environment.
$HOME/.profile	New user's profile, read by login shell to set up environment.
/var/adm/sulog	Log file for each attempt to su to a different user name.
$HOME/.cshrc	Read at beginning of execution by each shell.
$HOME/.login	Read by your login shell to set up environment.

APPLICATIONS

The **su** command is used to change to a new environment without losing your current shell environment. By using **su** you can become a different user while leaving your current environment alone. When you finish in the new environment you simply return to your previous shell environment.

Sometimes confusion arises about when to use **su** and when to use **login**. The **login** command execs a new shell, completely replacing the current environment and shell. The **su** command spawns a new shell, overlaying the current environment. If you use **su**, you can return to the previous environment. If you use **login**, you cannot return to the previous environment because it no longer exists.

Single shell commands can be passed to a different user's environment and executed with that user's user ID, group ID, and environment.

TYPICAL OPERATION

In this activity you use the **su** command to log in as yourself. Using the **env** command you will see how the environments are affected.

1. Type **export TEST_VAR="Some value"** and press **Return**.

C Shell

Type **setenv TEST_VAR_"Some value"** and press **Return**.

---

2. Type **env | col** and press **Return**. Make a mental note of your environment to compare to the next few illustrations. If convenient, print a copy by typing **env | col | lp** and pressing **Return**. Notice the value of TEST_VAR.

C Shell

Type **prinenv | col** and press **Return**.

---

3. Type **su mylogin** and press **Return**. Type your password and press **Return**. Notice the display on your screen; it resembles your initial login sequence.

```
cj> su mylogin
passwd:
cj>
```

4. Type **env | col** and press **Return**. Notice your environment is exactly the same as before. Particularly, the TEST_VAR value is unchanged.

C Shell

Type **printenv | col** and press **Return**.

---

5. Type **exit** or press **Ctrl-D** to exit **su** and return to your login shell.
6. Type **su - mylogin -c env** and press **Return**. Notice the output of the env command is displayed on your screen but you did not remain logged in as yourself. Because you used a - (hyphen) the environment was changed as if you had just logged in; no variables were passed to the **su** shell.

C Shell

Type **su - mylogin -c printenv** and press **Return**.

---

7. Turn to Module 45 to continue the learning sequence.

Module 128
sum

DESCRIPTION

The external **sum** command calculates a checksum and block count for a file.The checksum of a file is basically the sum of the values of all bytes stored in the file (or some variation of masking or shifting each byte). A checksum is useful for file validation. You can use **sum** to validate file transmission to a different system worked. Another use is to save a file to tape, reload the file, and rerun **sum** to compare that the version on the tape and the original files are the same.

Some versions of **sum** are not compatible between different versions of UNIX. Therefore, you should use **sum** when moving files between like architecture. You may use it between different architectures but should probably perform a simple test on a known ASCII file. For example, create a two- or three-line file (assume you use newfile as the filename). Execute

```
cj> sum newfile
46412     1
```

on the system you created the file. Transfer the file to a target system. Execute

```
hostx> sum newfile
46412     1
```

on the target system. If the checksums are the same, then the two system **sum** programs are probably compatible. Display the contents of the new file and visually compare the contents of the original and the new file.

COMMAND FORMAT Following is the general format of the **sum** command.

```
sum [ -r ] file_list
```

BSD (Berkeley)

```
sum file_list
```

Options The following describes the option and the argument that may be used to control how **sum** functions.

-r Forces **sum** to use a different algorithm to check the file.

BSD (Berkeley)

BSD does not support the -r option.

Arguments The following argument may be passed to the **sum** command.

file_list The name of one or more files to check the sum of all the bytes in each file.

DIAGNOSTICS AND BUGS If **sum** has a problem reading the file, the following message may be displayed.

```
Read error
```

This usually means **sum** tried to read past the end-of-file.

RELATED COMMANDS Refer to the **wc** command described in Module 154.

APPLICATIONS

You can use the **sum** command to checksum a file before you transmit it across communication lines. Then when it arrives you can checksum it on the destination system with **sum** to verify that it arrived without errors. This procedure is often used to verify release tapes that were saved and reloaded without read or write errors.

It may also be used to check for bad spots in a file. The system administrator may use it to check for tampering among system files.

TYPICAL OPERATION

In this activity you use the **sum** command to checksum a file before and after you copy it to a new directory. Begin at the shell prompt.

1. Type **sum /etc/passwd** and press **Return**. This will return a checksum for your system's passwd file.
2. Now copy the /etc/passwd file to your HOME directory by typing **cp /etc/passwd passwd** and pressing **Return**.
3. Now checksum the new file to verify the copy worked correctly by typing **sum passwd** and pressing **Return**. The number displayed should match the previous checksum returned earlier.
4. Turn to Module 154 to continue the learning sequence.

Module 129
suspend (csh, ksh)

DESCRIPTION

The internal **suspend** command is actually a keystroke and a command. The default keystroke is Ctrl-Z. The Ctrl-Z suspends the current process. The process is placed in a suspended state and interactive control returns to the parent shell. For instance, assume you start a long process, such as cd /; ls -lR > /tmp/ls.out, and forget to place the command in background. You can suspend it and then place it in background for further processing.

The Ctrl-Z is used to suspend any type of process that handles the suspend signal. It can be typed at any time. If the current process traps the suspend signal, the Ctrl-Z is processed and the job is suspended.

The **suspend** command can only be used while you are at a shell prompt. It is used to suspend a background job.

COMMAND FORMAT Following is the general format of the *suspend* command.

```
suspend
Ctrl-Z
```

RELATED COMMANDS Refer to the **bg**, **fg**, **jobs**, **kill**, and **stop** commands described in modules 9, 51, 68, 70, and 125.

APPLICATIONS

The **suspend** command provides a way to switch between multiple shells. You might have a login shell, a subshell in a different directory, and a **su** shell running simultaneously. To change between the three shells you can use the **suspend** command. You may need to use the **jobs** command. You must use the **fg** *%job* **or** bg *job* command to restart a suspended shell or process.

The Ctrl-Z key is useful when you are controlling processes within the same shell. You can toggle between **vi** and your shell using the Ctrl-Z key.

TYPICAL OPERATION

In this activity you use the **suspend** command to switch between two different shells. Begin at the shell prompt.

1. Type **PS1=memo-$PS1 ksh** and press **Return**. A new shell is spawned with a prompt of *memo-oldprompt*>.

2. Type **suspend** and press **Return** to suspend execution of the new subshell and return to your parent shell.

```
cj> suspend
[1] + Stopped (signal)        PS1=memo-$PS1 ksh
```

3. Type **jobs** to display the suspended shell.

```
cj> jobs
[1] + Stopped (signal)              PS1=memo-$PS1 ksh
```

4. Type **fg %1** to bring the suspended shell into the foreground. You can now interact with the subshell.
5. Type **Ctrl-D** to exit the subshell.
6. Turn to Module 125 to continue the learning sequence.

Module 130
tabs

DESCRIPTION

The external **tabs** command is used to set the tab stops on your terminal. The BSD version of **tabs** is limited compared to the System V version. You can use the **tabs** command to reset the tab stops on your terminal. The **tabs** program allows you to reset the tab stops using four different formats. These formats are:

- Arbitrary format; numbers define which columns have tab stops.
- Canned format; predefined tab stops are used.
- Repetitive format; tab stops are set every *n* columns.
- File format; tab stop format specifications are stored in a file.

Other options allow you to:

- Set the left margin of your text screen.
- Set the terminal type used to set tab stops. Overrides the TERM variable.

NOTE

Your terminal must support remote setting of hardware tab stops for **tabs** to work correctly.

COMMAND FORMAT Following is the general format of the **tabs** command.

```
tabs [ tabspec ] [ -Tterminal_type ] [ +mn ]
```

BSD (Berkeley)

```
tabs [ -n ] [ terminal_type ]
```

Options The following list describes the options and their arguments that may be used to control how **tabs** functions. The first set of options are different formats used to set tab stops.

Tabspec These options define different formats used to set tab stops on your terminal. This set of options is divided into four groups.

Arbitrary Settings Use arbitrary numbers to set tab stops where you desire.

n1,n2,n3,... Set tab stops in columns n1, n2, n3, and so on. All numbers must be separated by a comma. If you priced a number with + (plus sign) the number will be used as an increment added to the last tab stop number. Thus the two formats:

```
tabs 4,10,12,18,26
tabs 4,+6,+2,+6,+8
```

are equivalent.

Canned Settings Use one of the following codes to select a canned set of tab stops. Canned settings are predefined tab positions.

-a Set tab stops at columns 1, 10, 16, 36, and 72. Used for IBM S/370 Assembler coding format, first format.

-a2 Set tab stops at columns 1, 10, 16, 40, and 72. Used for IBM S/370 Assembler coding format, second format.

-c Set tab stops at columns 1, 8, 12, 16, 20, and 55. Used for normal COBOL format.

-c2 Set tab stops at columns 1, 6, 10, 14, and 49. Used for compact COBOL format. Columns 1-6 are omitted, thus the first character begins in columns 7.

-c3 Set tab stops at columns 1, 6, 10, 14, 18, 22, 26, 30, 34, 38, 42, 46, 50, 54, 58, 62, and 67. Used for compact COBOL format. Columns 1-6 are omitted. These are the recommended tab stops for COBOL.

-f Set tab stops at columns 1, 7, 11, 15, 19, and 23. Used for FORTRAN coding format.

-p Set tab stops at columns 1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, and 61. Used for PL/I coding format.

-s Set tab stops at columns 1, 10, and 55. Used for SNOBOL coding format.

-u Set tab stops at columns 1, 12, 20, and 44. Used for UNIVAC 1100 Assembler coding format.

Repetitive Settings This sets tab stops every *n* columns. This is how the normal tab stops are set.

-*n* Set tab stops every *n* columns. The first tab stop is at 1+*n*. The second stop is at 1+(n*2), the third is at 1+(n*3), and so on. The UNIX system "standard" is based on a repetition of 8. This places stops at 9, 17, 25, etc. The value of 0 may be used to specify no tab stops.

Setting stored in Files Tab stops may be defined and stored in a file that can later be used to guarantee tab settings are correct before a file is displayed.

-- file The first line of the file must contain a format specification as described in section 4 of the UNIX Programmers Reference Manual.

Other Options The following are the remaining options used with the **tabs** command.

-T*terminal_type* Specify the terminal_type. **tabs** usually needs to know the type of terminal to send the correct tab setting sequence to the terminal. The terminal type must be known to send margin settings. *Terminal_type* is valid terminal type stored in the *terminfo* (or /etc/termcap) database.

If -T is not specified, **tabs** uses the value of your environment variable TERM. If TERM is not defined, **tabs** sends a generic sequence to set the tabs that works for multiple terminal types.

+m*n* Set left margin on the terminal. The terminal type must be known by **tabs** for the margin to be set. Setting the margin causes the left side of the text display to move over to column 1+*n*. If no *n* is provided, **tabs** assumes a number of 10 and the left margin is set to column 11. To reset your margin to normal, the leftmost setting, use a value of 0 for *n*. The GE TermiNet terminal must have the left margin specified as 1.

BSD (Berkeley)

-n The left margin of the terminal is not indented.

Arguments The following list describes the arguments that may be passed to the **tabs** command.

BSD (Berkeley)

terminal_type The name of a terminal type. The terminal type must be a valid terminal found in the /etc/termcap file.

If no *terminal_type* is provided, a default setting is used. This is a fairly standard setting and works for many of the older "dumb" terminals.

FURTHER DISCUSSION When you press the Tab key on your terminal a tab character (Ctrl-I) is sent to the terminal and on to UNIX. The terminal moves the cursor to the next "tab stop" set on your terminal. The normal setting for tab stops is every eighth column. This equates to columns 9, 17, 25, 33, 41, and so on. By using **tabs** you can reset these tab stops to new column positions.

You can also set margins on terminals that support margin control. By default, **tabs** uses the value of the TERM variable. For proper functioning of **tabs** you should define the TERM variable or use the -T option. The -T option overrides the TERM value.

DIAGNOSTICS AND BUGS Some terminals may interpret the first tab setting as a margin setting. Others may not clear tabs as requested. The **tabs** command can only clear up to 20 tab stops but can set up to 64.

The following messages may be returned by **tabs** if a problem is incurred:

illegal tabs	Arbitrary tabs are not in the correct order.
illegal increment	Arbitrary tab numbers contain a 0 or missing increment number.
unknown tab code	A code used for a canned tab setting does not exist.
can't open	A file cannot be opened. If you used the --file option, then file does not exist.
file indirection	The file referred to by --file contains a reference to yet another file.

RELATED COMMANDS Refer to the **stty** and **tput** commands described in modules 126 and 139.

It may be necessary to read *fspec* and *terminfo* entries in section 4 of your *UNIX Programmers Reference Manuals* to fully understand how **tabs** works.

RELATED FILES The **tabs** command writes to the standard output to send its character sequences to your terminal to set the tab stops and margins.

APPLICATIONS

The primary use of the **tabs** command is to provide a formatted screen using tabs. This makes output more readable. It is especially useful for programming languages. You can set your tab stops to every four spaces on the screen, but when you print the listing you can have **pr** expanding your tabs to every 8 spaces. Thus you can compact your code on-line (on your screen) and expand it to a more readable format on the printout.

TYPICAL OPERATION

In this activity you use the **tabs** command to reset your terminal's tab stops to every fourth column. Begin at the shell prompt.

1. Type **tabs -4** and press **Return**. Now your tab stops on your terminal are set to every fourth column. Some terminals only support tab stops at certain positions. If this is true the **tabs** command has no effect on your tab settings.
2. Try pressing the **tab** key multiple times. Your cursor should move across your terminal screen four spaces at a time. If it does not, your terminal may not support remote setting of hardware tab stops.
3. Type **tabs 2,4,6,8,10,14,18,22,26** and press **Return**. This sets your tab stops every two columns up to column 10, then it sets them to every fourth column up to column 26. This may be useful for displaying shell scripts that normally wrap around your screen.
4. To reset your tab stops to every 8 columns, type **tabs** and press **Return**.
5. Turn to Module 19 (SV), Module 46 (BSD) to continue the learning sequence.

Module 131
tail

DESCRIPTION

The external **tail** command displays the last few lines of a file. It displays however many lines you specify as an option. If you do not specify a number of lines, **tail** displays the last ten lines of a file. Its cousin is the **head** command.

The **tail** command also provides an automatic displaying mode. You may have a process that generates output to a log file and you want to watch the log as the process executes. By using the -f option, **tail** displays each new line in the file. This provides you the flexibility to watch the file when you want to for however long you desire.

COMMAND FORMAT The general format of the **tail** command follows.

```
tail [ -[n]bcfl ] file_list
tail [ +[n]bcfl ] file_list
tail [ -[n]bcrl ] file_list
tail [ +[n]bcrl ] file_list
```

BSD (Berkeley)

```
tail [ -[n]bcflr ] file_list
tail [ +[n]bcflr ] file_list
```

Options The following list describes the options and their arguments that may be used to control how **tail** functions.

-n — The number of lines **tail** will display from the end of the file. If *n* is not provided, **tail** displays the last ten lines of the file.

+n — The number of lines **tail** will skip from the beginning of the file before it starts to display lines. If *n* is not given, **tail** skips ten lines.

*-n*b
*+n*b — Display *n* blocks of text instead of *n* lines.

*-n*c
*+n*c — Display *n* characters of text instead of *n* lines.

*-n*l
*+n*l — Display *n* lines of text. This is the default if no text type option is provided.

*-n*f
*+n*f — Follow the growth of a file. After the **tail** command has read and displayed the requested lines of data it enters an endless loop. It waits one second then checks for additional lines added to the file. If new lines have been added by another process, then it displays them. It continues this sleep and check process until

you terminate it. If you are executing **tail** interactively, the interrupt signal causes termination. If **tail** is running in the background, then use the **kill** command to terminate it. This option allows you to monitor the growth of a file being written to by another process. The file **tail** is reading must not be a pipe.

*-n*r
*+n*r — Displays lines from the end of the file in reverse order. The last line of the file is displayed first. If you do not specify *n*, the entire file is displayed.

Arguments The following argument may be passed to the **tail** command.

file_list — A list of files that **tail** will display the last *n* (10) lines of to your screen.

DIAGNOSTICS AND BUGS If the *-n* option is used, **tail** stores the text of the file in a buffer until the end-of-file is read. It can then display the requested amount of lines relative to the last line. Because it buffers the data, **tail** is limited to certain file sizes.

If you use **tail** with character special files it may behave strangely.

RELATED COMMANDS Refer to the **cat**, **head**, and **pg** commands described in modules 13, 63, and 109.

RELATED FILES The **tail** command can read from the standard input or files. It writes to the standard output.

APPLICATIONS

The **tail** command is useful when you only need to view the last few lines of a large file. It can be used to see if ASCII files were completed correctly by another process. For example, if you transmit a file across telephone lines, you can use **tail** to see if the file ends correctly.

It can also be used to monitor the progress of another process. If the other process is writing output to a file, **tail** with the -f option can be used to display the lines of the file as the other process writes them.

TYPICAL OPERATION

In this activity you use the **tail** command to display the last five lines of a file and all lines after line 4. Begin at the shell prompt.

1. Type **tail -5 file2** and press **Return**. Notice the last 5 lines of the file2 file is displayed.
2. Type **tail +5 file2** and press **Return**. **tail** skips to line five before it begins to display the contents. Your screen should look like the following display:

```
cj> tail +5 file2
        Just a few control Keys for demonstration.
=====
This text is being read by cat from my terminal keyboard.
Cat will read the text as it is typed and create a file named file1.

With a blank third line!
        Just a few control keys for demonstration.
```

3. Turn to Module 103 (SV), Module 90 (BSD) to continue the learning sequence.

Module 132
talk

DESCRIPTION

The external **talk** command lets you talk to another user. You "call" a user with talk. Once the other user responds by using **talk** *user* the conversation is established. Each character you type is displayed on the top half of your screen and the bottom half of the other user's screen. As the other user replies, his characters are displayed on the bottom half of your screen.

To discontinue the conversation simply type a Ctrl-D at the beginning of a line. To find out who is logged on for you to talk with, use the **who** command.

COMMAND FORMAT The format of the **talk** command follows.

```
talk username [ ttyname ]
```

Arguments The following list describes the arguments that may be passed to the **talk** command.

username — The login name of the user you wish to communicate with via **talk**. If the system you are using is connected to a network, *username* may have a different format. See the following section, Calling other Users.

ttyname — The tty device name of the user you wish to talk with using **talk**. If the same user is logged in more than once, you must distinguish which of the login sessions you want to communicate with. Use the **who** command to display a list of users who are logged in to the system. If necessary, add the tty number to the command line for **talk**. The ttyname format is /dev/tty*XX*.

FURTHER DISCUSSION Once the communications link has been established you can use the following special characters to control the screen.

Ctrl-L	Redraw the screen. Useful if it gets garbled.
intr	Interrupt the talk session. Causes immediate termination.
erase	Erases the previous character. The **stty erase** character.
kill	Erases the entire current line. The **stty kill** character.

BSD (Berkeley)

werase	Erases the previous word. The **stty werase** character.

The **stty** command is used to set the **erase**, **kill**, and **werase** characters.

Permission to talk A user may deny incoming communications by using the **mesg** command. If a user has set message receiving to no, then you are not allowed to **write** or **talk** to the user. Certain commands, **vi** and **pr**, do not allow communications while they are executing.

Calling other users The following table describes methods for calling other users.

user	The name of a user on the local system you are logged in on.
host.user *host:user* *host!user* *user@host*	Various formats for contacting a user on a remote system. The most desirable format is *user@host*, which is the Internet address format.

NOTE

If the system you use is not connected to a network, you do not need to use the host notation when calling other users.

Example The following screens show an example of using **talk**.

Your terminal (username = mylogin)

```
cj> who
mylogin tty005     Aug 20  9:05
jack    tty019     Aug 20 10:21
jill    ttyp01     Aug 20  8:14
jimmy   ttyp03     Aug 20  7:53
cj> talk jill
[No connection yet]
```

changes to

```
[Waiting for your party to respond on tty005]
```

At jill's terminal.

```
cj>
Message from mylogin
talk: connection requested by mylogin
talk: respond with: talk mylogin
talk luwis
```

Your screen changes to:

```
----------------------------[Talking to jill on tty005]----------------
```

At this point you have contacted jill and she has decided to respond to your talk request. Now your screen is cleared and divided in half. The top is your talking half and the bottom is your listening half.

If jill is slow to respond, you receive the following message:

```
[Ringing your party again on tty005 (1st retry)]
```

This may occur three times before **talk** gives up and displays:

```
[Your party would not respond]
```

RELATED COMMANDS Refer to the **mail**, **mesg**, **who** and **write** commands described in modules 86, 88, 159, and 161.

RELATED FILES The **talk** command uses the /etc/hosts file to locate the recipient's host system's address on a network. The /etc/utmp file is accessed by **talk** to locate the recipient's tty.

APPLICATIONS

The **talk** command is useful to carry on conversations between two users. You can visually **talk** to another user to exchange information immediately. Both users can type simultaneously thus speeding up the conversation compared to the line-oriented **write** command. Although **talk** is not as personal as verbal communications, it allows users to communicate without having to be within hearing range.

In some environments one user may be in Texas while the other is in California. Another benefit of **talk** is to provide examples of text to a user. If you have ever tried to step someone through typing in code or even a command line, you know how difficult it may be to convey your message. By using **talk** you can simply type the code or command and have the other user look at it.

TYPICAL OPERATION

In this activity you use the **talk** command to communicate with another user on your system. You must first find someone you know who will talk back via **talk**. But this usually is not too difficult. Begin at the shell prompt.

1. Type **who** and press **Return** to list who is currently logged on to the system. Select a user you wish to talk to via your terminal.
2. Type **talk** *user* and press **Return** where user is the user you selected to talk to on your computer. Now wait for the user to respond. If the user does not respond within a given time (usually 30 seconds), **talk** will inform you that the user is not responding and stop trying to call.
3. Once the user has responded you can begin typing as you wish.
4. When you are ready to stop talking, send the other user a message such as, **Nice talking, but I've got to get back to work!** and press **Return**.
5. Now press **Ctrl-D** to disconnect from talk.
6. Turn to Module 47 to continue the learning sequence.

Module 133
tar

DESCRIPTION

The external **tar** command is a tape archive program. It is used to archive files from disk to tape. The media does not have to be tape. You can create a tar file on disk, diskette, cartridge tape, or 9-track tape. You also use **tar** to retrieve archived files from a tar file.

COMMAND FORMAT Following is the general format of the **tar** command.

```
tar -c[bfLvw[ds]]    block device filelist
tar -r[bLvw[ds]]     block device [ filelist ]
tar -t[fLv[ds]]      device
tar -u[bLvw[ds]]     block device [ filelist ]
tar -x[flLmovw[ds]]  device [ filelist ]
```

BSD (Berkeley)

```
tar -c[Bbfhvw[d]]    block device  filelist -C alt_files
tar -r[Bbhvw[d]]     block device [ filelist ] -C alt_files
tar -t[fv[d]]        device
tar -u[Bbhvw[d]]     block device [ filelist ] -C alt_files
tar -x[Bflmopvw[d]]  device [ filelist ]
```

NOTE

Some BSD systems do not support the hyphen (-) preceding the options. Thus it must be left off the command line for **tar** to work properly.

Options The following list describes the options and their arguments that may be used to control how **tar** functions. There are two types of options used with **tar**. There are options that control how **tar** functions and options that modify how each function performs.

FUNCTION TYPE OPTIONS These options specify what function **tar** is to perform.

-c Creates a new archive file. The writing starts at the beginning of the archive. The -r option is automatically assumed.

-r Replaces the named *files*. The *files* are written on the end of the tar file.

-t Displays a table of contents of all files contained in the tar file.

-u Updates the archive file. Adds the *files* to the tar file if they are not already archived and if they have been modified since the last archive was written.

-x Extracts *files* from the tar file. If no *files* are listed, all files are extracted from the tar file. If multiple versions of a file exist in the tar file, only the last (latest) version is extracted. If the named *file* is a directory, then the entire directory tree

is extracted. If a file no longer exists on disk, the file is created with the modes from the archived version of the file. The set-UID and set-GID bits are not set on extracted files unless you are super-user.

MODIFIER TYPE OPTIONS These options modify how **tar** performs the specified function.

-b Causes **tar** to set the blocking factor to the size specified in the *block* argument. The value of *block* can be from 1 to 64. The default is 1. This modifier should only be used when you wish to extract files from or write files to a raw character special archive. The block size is automatically determined when extracting files from a block special device.

-f Causes **tar** to use the *device* argument as the output device where the tar file is written. If the *device* is - (hyphen) **tar** writes to the standard output or reads from the standard input, depending on the function options specified. This allows you to use **tar** in a pipeline command.

ds This option determines which tape drive (*d*) and what speed (*s*) to use for the archive. This notation is used on larger systems where multiple tape drives exist. A typical set of devices might be /dev/mt/0m through /dev/mt/5m. The *d* option specifies 0 through 5. The *m* describes medium speed(1600 BPI). Tape drives may be any media device. The default device is 0m (/dev/mt/0m).

-l Causes **tar** to display messages for all links it cannot resolve during archival of files. If -l is not used, no error messages are displayed. This option is only useful with the -c, -r, and -u options.

-L Causes symbolic links to be followed.

-m Causes **tar** not to restore modification times to the time held before archival. Instead, the modification time of each file is set to the time the file is extracted.

-o Causes **tar** to set the user ID and group ID to your IDs instead of the IDs saved on tape. This option is only useful with the -x option.

-v Causes **tar** to be verbose. It displays the function and filename for each file being archived or extracted.

-w Causes **tar** to perform the verbose option and after each function and filename is displayed, it waits for you to confirm whether to perform the function or not. If you start a string with y, the function is performed, otherwise the filename is skipped in the process.

BSD (Berkeley)

-B Forces input and output blocking to 20 blocks per record.

-b Same as System V, except the default blocking factor is 20. The default blocking factors varies from system to system and depends on the device being written to by tar.

-C Change directory to the following directory argument.

-h Forces **tar** to treat symbolic links as normal files or directories. Normally symbolic links are not followed.

-p Restores files to their present mode. Your current umask is ignored. If you are the super-user, the set-uid and sticky bit information are also extracted. Only usable with the -x option.

0-9 Specifies the device used to access the tape drive. The base of the device name may vary among systems. Common defaults are /dev/rmt[0-9], /dev/rst[0-9], /dev/rct[0-9]. The number may be two digits.

Arguments The following list describes the arguments that may be passed to the **tar** command.

block Specifies the blocking factor for raw devices. The default is set to 1. The possible block sizes range from 1 to 64.

device The name of the device to be used for the archive. Each system usually has multiple device drivers for each physical media device. The following table lists some of the more popular drives.

D (Drive)	*Description*
/dev/rmt/0m	Raw magnetic tape, 1600 bpi.
/dev/rmt/0h	Raw magnetic tape, 6250 bpi.
/dev/mt/0m	Magnetic tape, 1600 bpi.
/dev/mt/0h	Magnetic tape, 6250 bpi.
/dev/mt/ctape	Cartridge tape.
/dev/ctape	Cartridge tape.
/dev/diskette	Floppy diskette.

The possible speeds for 9-track tapes are:

S (Speed)	BPI (Bytes per inch) Rating
l	800 BPI
m	1600 BPI
h	6250 BPI

Use these speed letters to replace the *s* in the *ds* option.

BSD (Berkeley)

D (Drive)	*Description*
/dev/rmt0	Raw magnetic tape, the default drive. (rmt8)
/dev/rmt8	Raw magnetic tape, 6250 bpi, rewind.
/dev/rmt12	Raw magnetic tape, 6250 bpi, no rewind.
/dev/rmt16	Raw magnetic tape, 6250 bpi, rewind.
/dev/rmt20	Raw magnetic tape, 6250 bpi, no rewind.
/dev/rst0	Raw SCSI cartridge drive. Usually rst8.
/dev/rst4	Raw SCSI. QIC 02 format.
/dev/rst8	Raw SCSI. QIC 24 format.
/dev/rct0	Raw cartridge tape.
/dev/rct4	Raw QIC 02 cartridge tape.
/dev/rct8	Raw QIC 24 cartridge tape.

---

filelist The list of filenames and/or directories you wish to archive or unarchive (extract).

EXAMPLES Some examples of how to use **tar** follow.

1. Archiving more than one directory. To archive more than one directory you can use the -C option of BSD. For System V you just specify the list of directories. The -C allows short relative pathnames. The following example is for a BSD **tar**. It copied the *bin* subdirectory and the sibling directory of *jim/bin* to tape.

```
cj> tar cf -C ./bin -C ../jim/bin /dev/rmt8
```

2. Copying one directory structure to a new directory location on disk.

```
cj> cd fromdir
cj> tar cf - . | ( cd todir; tar xf - )
```

The first **cd** command changes your working directory to where you want to copy from, such as your HOME directory. The tar command creates a tar file to the standard output. The output is piped to a subshell which changes directories (**cd**) to the directory you want to copy the files to; for example, a new HOME directory on a different file system. The second **tar** command extracts the files back out into a hierarchical structure.

3. Archiving the current directory. This example shows how to save your home directory to tape on a System V UNIX machine.

```
cj> tar -cvf /dev/rmt/0m .
```

4. Listing contents of an archive tape. The following example lists the contents stored on tape from a previous **tar** command on a BSD UNIX machine.

```
cj> tar -tvf /dev/rmt8 .
rwxr-xr-x 901/100        1321      Aug 19 15:54:1989   ./
rwxr-xr-x 901/100        1321      Aug 19 15:54:1989   ./bin/
.
.
.
```

The format is,

```
permissions uid/gid      size      date    time        file
```

5. Extracting files from an archive tape. The following example assumes you are on a System V system.

```
cj> cd desired_directory
cj> tar -xvf /dev/rmt/0m
x ./file1, 199 bytes, 1 tape blocks
.
.
.
```

6. The find command will locate all files newer than the .lastsave file and print the pathname. The output of the **find** command is sent to the **tar** command and the files are added to the *arc* archive file. You must create the .lastsave file the first time you run the **find** command. Each time you run **find** you must **touch .lastsave** so **find** will have a time to compare against for the next save.

```
cj> touch 11160101 .lastsave # set date back a few days Nov 11 1:00AM
cj> find . -newer .lastsave -print | tar -rf arc -
cj> touch .lastsave
```

DIAGNOSTICS AND BUGS The **tar** command does have its shortcomings. The following list provides a brief description of those features that would be nice to have.

There is no way to recover a specific version of a file.
If a tape error occurs **tar** usually aborts. Error recovery would help.
The -u option is often extremely slow.
The -b option should not be used if later updates are to be applied.
Empty directories and special files are not archived.
The magnetic tape device driver cannot perform backspaces on raw magnetic tapes.

RELATED COMMANDS Refer to the **cpio** command described in Module 25.

RELATED FILES The **tar** command uses a temporary file to perform its processing.

/tmp/tar* The temporary work file **tar** uses to perform its task.

APPLICATIONS

The most common use of **tar** is to create archive tapes and extract files when needed. The **tar** command is more sophisticated than the **cpio** program. It can update the existing archive file without having to rewrite the file from the beginning. It can also be used to copy directory structures from one directory to a new directory.

TYPICAL OPERATION

In this activity you use the **tar** command to archive a directory structure to a temporary file. If you can arrange to use the tape drive on your system, you might want to try this activity using the device name of your system's tape drive. But for simplicity we use a file on the system to archive your files. Begin at the shell prompt in your HOME directory.

1. Type **tar -cvf /tmp/mylogin.arc .** and press **Return**. This command archives your current directory creating an archive file named /tmp/mylogin.arc.
2. Type **col < /tmp/mylogin.arc | pg** and press **Return**. Notice the file is in a readable format. This is the archive file you just created. If binary or data files have been archived, your display may be garbled and/or your terminal may hang. That is why you use the **col** command.
3. Type **q** and press **Return** to exit **pg**.
4. Type **tar -tvf /tmp/mylogin.arc** and press **Return** to list the filenames of each file contained in the archive file.
5. Turn to Module 31 to continue the learning sequence.

Module 134
tee

01/DECEMBER/2014
10:20

DESCRIPTION

The external **tee** command is used to write to the standard output and to a file simultaneously. You place the **tee** command anywhere in a pipe command to divert a copy of the standard input (of **tee**) to disk and another copy to your terminal. Thus its name is from the world of plumbing; it allows one input and two outputs. The **tee** command actually allows multiple file copies. For example,

```
sort unsorted | tee sorted | cut -c-20,41- | tee cutup | lp
```

sorts the data in the file *unsorted*; **tee** then sends a copy to the file **sort**ed. The standard output of **tee** is piped to **cut** which removes columns 21 through 40 and pipes it to another **tee**. The second **tee** writes a copy to the file *cutup* and pipes a copy to **lp** for printing.

COMMAND FORMAT Following is the general format of the **tee** command.

```
tee [ -ai ] file_list
```

Options The following options may be used to control how **tee** functions.

-a — Appends the output to an existing file. If the file does not exist it will be created. Normally an existing file would be overwritten.

-i — Ignore interrupts. If you press the Delete key, **tee** ignores the interrupt signal sent to it and continues processing.

Arguments The following argument may be passed to the **tee** command.

file_list — One or more files where **tee** writes copies of the input.

RELATED FILES The **tee** command reads from the standard input and writes to the standard output and any files listed on the command line.

APPLICATIONS

You can use the **tee** command to execute a command interactively (in foreground) and save a copy of the output on disk. The **tee** command can be placed anywhere in a pipe to check what the output looks like at a certain point. Thus it lends itself to debugging pipe commands and capturing output for later perusal.

TYPICAL OPERATION

In this activity you use the **tee** command to receive input from the sort command, send the output to the printer, and store a copy in a file. Begin at the shell prompt.

1. Type **nawk -F: '{ print $1}' /etc/passwd | sort | tee users | lp** and press **Return**. Notice the message returned by **lp**. You now have a printout on the default printer. The output is also stored in the file named *users*.

BSD (Berkeley)

Type **nawk -F: '{ print $1}' /etc/passwd | sort | tee users | lpr** and press **Return**.

---

2. Type **cat users** and press **Return** to display the contents of the *users* file.
3. Turn to Module 116 (BSD) to continue the learning sequence.

Module 135
test

DESCRIPTION

The internal **test** command is used to evaluate conditional expressions. The **test** command evaluates the expression and, if it is true, returns a 0 return code; if the expression is false, the return code is 1. The test command is primarily used to make a conditional decision in a shell script. The following features are provided:

- Return code used by shell constructs; if, until, and while.
- Evaluates conditional (true or false) expressions.
- Returns an exit status equal to 0 if condition is true.
- Returns an exit status greater than 0 if condition is false.
- Evaluates file permissions and types.
- Evaluates numerical expressions.
- Evaluates string expressions.

There are internal versions of test in the csh and ksh and an external version in /bin/test. This module explains the ksh version and the common BSD /bin/test version.

COMMAND FORMAT Following is the general format of the **test** command.

```
test expr
[ expr ]
```

The brackets ([]) are an alternative to the **test** syntax. You must place blanks before and after the first bracket and before the second one. Thus the following two syntaxes are equivalent:

```
if test -f file
if [ -f file ]
```

EXPRESSIONS The **test** command has primary expressions (operators) that perform one specific conditional check. These primary expressions can be combined using operators. The following three sections cover the primary expressions used to test files, strings, and numbers.

Testing on files These primitives can be used to test the state, type, or permissions of a file. The *file* may be a constant or variable.

NOTE

If a file does not exist or the condition is false, **test** returns an exist status of nonzero. If the condition is true, **test** returns an exit status of 0.

Expression	*Description*
-b *file*	True if file exists and is a block special file.
-c *file*	True if file exists and is a character special file.
-d *file*	True if file exists and is a directory.
-f *file*	True if file exists and is a regular file.
-g *file*	True if file exists and has the set-group-ID bit set.
-k *file*	True if file exists and has the sticky bit set.
-p *file*	True if file exists and is a pipe.
-r *file*	True if file exists and is readable.
-s *file*	True if file exists and has a size greater than 0 bytes.
-t [fds]	True if the open file defined by file descriptor "fds" is a tty(default fds=1).
-u *file*	True if file exists and has the set-user-ID bit set.
-w *file*	True if file exists and is writable.
-x *file*	True if file exists and is executable.

Korn Shell Extensions

-L *file*	True if file exists and is a symbolic link.
file1 -nt *file2*	True if *file1* is newer than *file2*.
file1 -ot *file2*	True if *file1* is older than *file2*.
file1 -ef *file2*	True if *file1* has the same device and inode number as *file2*.

If file is of the form /dev/fd/*n*, where *n* is an integer, the test is applied to the open file whose descriptor number is *n*.

BSD (Berkeley)

Only the following file testing options exist in BSD versions.

-L *file*	True if file exists and is a soft link.
-d *file*	True if file exists and is a directory.
-f *file*	True if file exists and is a regular file.
-r *file*	True if file exists and is readable.
-s *file*	True if file exists and has a size greater than 0 bytes.
-t [fds]	True if the open file defined by file descriptor "fds" is a tty(default fds=1).
-w *file*	True if file exists and is writable.

NOTE

It is advisable to enclose string variables within double quotes. This ensures that **test** will work even if the variable is unset or null.

Testing on strings The following primitives can be used to test the condition of strings. The *sX* values may be a constant or a variable. The double quotes around the *sX* values are not necessary but are advisable.

Expression	*Description*
-n "*s1*"	True if length of string *s1* is not zero.
-z "*s1*"	True if length of string *s1* is zero.
"*s1*" = "*s2*"	True if strings *s1* and *s2* are equal(match).
"*s1*" != "*s2*"	True if strings *s1* and *s2* are NOT equal.
"*s1*"	True if *s1* exists; is not a null string.

The = and != operators have higher precedence than the -b through -z operators.

BSD (Berkeley)

BSD supports all of these options.

Testing on numbers The following primitives can be used to test the condition of two numbers. The double quotes around the *sX* values are not necessary but are advisable.

Expression	*Description*
n1 -eq *n2*	True if *n1* and *n2* are equal.
n1 -ne *n2*	True if *n1* and *n2* are NOT equal.
n1 -gt *n2*	True if *n1* is greater than *n2*.
n1 -ge *n2*	True if *n1* is greater than or equal to *n2*.
n1 -lt *n2*	True if *n1* is less than *n2*.
n1 -le *n2*	True if *n1* is less than or equal to *n2*.

BSD (Berkeley)

BSD supports all of these options.

OPERATORS The following operators may be placed between expressions, except for the parentheses, which are placed around expressions for grouping.

Expression	*Description*
! *expr*	Unary *negation* operator. Unary means it only negates one expression, the one immediately following it. To negate multiple expressions enclose them in parentheses. For example. `[ ! -w myfile ]` returns true if *myfile* does not exists or is not writable. While `[ ! \( -r myfile -a -w myfile \) ]` returns true if *myfile* does not exist or is not readable and writable.
expr -a *expr*	Binary *and* operator. Binary means it operates on two expressions, the one before and the one after the -a. The result is true if both expressions are true. If one or both expressions are false, then the result of the test is false. For example, `[ -r myfile -a -s myfile ]`

returns true if you can read *myfile* and it is not empty.

expr -o *expr* — Binary *or* operator. The result is true if one or both of the expressions are true. If both expressions are false, then the result of the test is false. For instance,

```
[ "$ANS" = "Y" -o "$ANS" = "X" ]
```

returns true if the value of $ANS equals Y or X.

(*expr*) — Expression grouping parentheses. You may need to group expressions when using operators to force the correct interpretation of the operators. The **test** command evaluates with the following precedence, from high to low:

```
(!), !, -a, -o
```

For example,

```
[ ! -r myfile -a -x myfile ]
```

returns true if *myfile* is not readable and is executable. Whereas,

```
[ ! \( -r myfile -a -x myfile \) ]
```

returns true if *myfile* is not readable, not writable, or both.

BSD (Berkeley)

BSD supports all of these operators.

FURTHER DISCUSSION The **test** command is normally used in shell scripts to test expressions and return an exit status to the **if**, **until**, and **while** constructs.

When using variables in the **test** command you should always surround the variable with double quotes unless they are being compared as numbers. For instance, if you had the variable FILE, you would type

```
if [ "${FILE}" = "myfile" ]
```

This syntax will prevent the test command from aborting if the variable is not set or has a NULL value.

DIAGNOSTICS AND BUGS If you own a file and test for a condition of -r, -w, or -x, you may not receive the correct results. If the file permissions are not set in the owner field for the condition being tested, the result of the test will be false. In short, the **test** command looks at the permissions that relate to you using the file.

RELATED COMMANDS Refer to the **ksh** command described in Module 71.

RETURN CODES The **test** command returns a 0 if the expressions evaluated were true. If the expressions where false then a 1 is returned.

APPLICATIONS

You can use **test** in shell scripts to test for file permissions and to compare the values of variables. Normally, you use **test** in conjunction with the programming constructs of the shell. For example,

```
cj> while [ -n "$*" ]
do
 echo $1
 shift
done
```

would test for the existence of parameters. When no more parameters existed the loop would exit because **test** returns a false (return code 1) value.

TYPICAL OPERATION

In this activity you use the **test** command to make sure a file is readable before you try to read it. Begin at the shell prompt.

1. Create the following shell script using an editor or by using the **cat** command. To use **cat**, type **cat > mycat** and press **Return**. Then type each line of text followed by a **Return**. Press **Ctrl-D** after you have finished typing all of the code.

```
cj> cat > mycat
if [ -r "$1" ]
then
   cat $1
else
   echo "You Do Not Have Read Permission for $1"
fi
^D
```

C Shell

```
cj> cat > mycat
#1/bin/csh
test -r $1
if ( $status == 0 ) then
   cat $1
else
   echo "You Do Not Have Read Permission for $1"
endif
^D
```

---

2. Now change the modes so you can execute your new **mycat** command by typing **chmod 755 mycat** and pressing **Return**.
3. To test your new shell script command, type **mycat xyz** and press **Return**. Notice the **echo** message should return. The script does not handle this correctly. It should test for the existence of the file, then check if it is readable.
4. Now try it with a filename that exists. Type **mycat /etc/passwd** and press **Return**. The password file should be displayed on your screen.
5. Turn to Module 141 to continue the learning sequence.

Module 136
time/timex

DESCRIPTION

The external **timex** command reports the time it takes to execute a command. It reports the elapsed wall clock time, the user time, and the system time. It can also be used to report accounting information. By using the **timex** command you can:

- Report execution time of a command (elapsed, user, and system).
- Report total number of blocks read and written.
- Report total system activity during command execution.
- Report various accounting information.

The **time** command is a pre-System V Release 3.0 utility that does not have any options. It only returns the elapsed wall clock, user, and system times for execution.

BSD (Berkeley)

The BSD version of UNIX does not support the **timex** command. There are two **time** commands available on BSD. One resides in /bin/time, the other is a C shell internal command. To execute the /bin/time you must specify the entire path if you are using the **csh**. If you are in a Bourne shell you do not have to specify the entire path.

C Shell

The C shell supports an internal **time** command. Refer to Module 137 on **time/times** for further information.

COMMAND FORMAT

Following is the general format of the **timex** and **time** command.

```
timex [ -ops ] command
time command
```

C Shell

From within the csh you must specify the entire path to run the Berkeley time command instead of the csh time command.

```
/bin/time command
```

Options The following list describes the options and their arguments that may be used to control how **timex** functions. Process accounting has to be running for some of the following commands to work properly.

-o Reports the total number of blocks read and written by the command and all child processes.

-p Report accounting records for the *command* and all child processes. The -p option has several suboptions to control which accounting records are reported. The suboptions are:

- -f Report the system exit status in the output and the *exec/fork* flag.
- -h Report the hog factor (CPU time/elapsed time). This is the fraction of total CPU time used by *command*.
- -k Report total kcore minutes, instead of memory size used.
- -m Report the mean core size. This is the normal reported output.
- -r Report CPU factor (user time/(system time + user time)).
- -t Report the number of blocks read and written and the system and CPU times.

-s Report all system activity during the execution of *command*. All data items in the **sar** report are displayed. The **sar** command exists on most System V systems. It is the System Activity Report generator. It is not considered a user command, rather an administrative command.

Arguments The following argument may be passed to the **timex** command.

command A UNIX command. Any executable program with options and arguments if they exist.

DIAGNOSTICS AND BUGS The **timex** command may report confusing results since it may include statistics for a background process executing with the same user ID, terminal ID, and execution window.

RELATED COMMANDS Refer to the **times** command described in Module 139.

RELATED FILES The **timex** command reads accounting information from the accounting file /usr/adm/pacct if the -p option is specified.

APPLICATIONS

The **timex** command is useful in timing the execution of commands. It is a necessary command if you are working on minimizing the execution time of a program. It can be used to track how much resources a process uses. It is sometimes added to the passwd file to time each user's login shell. Thus when you log out you will receive a report of statistics on your login session.

TYPICAL OPERATION

In this activity you use the **timex** command to time the execution of a UNIX utility. Begin at the shell prompt.

1. Type the following lines, pressing **Return** at the end of each line. After you finish the last line wait for the result. Depending on your computer's speed it may take a few seconds or minutes.

```
cj> timex bc <<!
2 ^ 9999
!
```

The output is about two 24 line screens of numbers. The amount of time it took your computer to calculate 2 to the 9999th power will also be displayed after the number. (The fastest I've seen is 17 seconds on a Mips workstation, unless you include a vectorized calculation by a Convex or Cray).

C Shell

```
cj> /bin/time bc <<!
2 ^ 9999
!
```

---

2. Turn to Module 137 to continue the learning sequence.

Module 137
time (csh) / times (sh, ksh)

DESCRIPTION

The internal **times** command reports the cumulative user and system times. The times reported are for the shell and all processes executed from the shell.

C Shell

The internal csh **time** command is slightly different from the sh/ksh **times** command. If a numeric argument is supplied, then the statistics of any program that uses more than this time in seconds are displayed. The statistics are the user, system, utilization percentage, and real times that it required to complete the process. The utilization percentage is derived from adding the user and system time and dividing by the real time.

The output of **times** is displayed in two columns and two rows. The information for the shell is displayed in row one. The cumulative times for all processes run under the shell are displayed in the second row. The first column represents the user time. The second column represents the system time used. The following display is an example of the output generated by **times**:

```
cj> times
0m02.25s        3m48.48s
0m33.50s        2m43.34s
```

The top left number represents 0 minutes and 2.25 seconds of user time have been used by the shell. The top right number shows that 3 minutes and 48.48 seconds have been used by the shell in system time. The next row reports the time used by all processes run under the shell.

C Shell

The output of the csh **time** command is as follows:

```
0.03u 0.10s 0:00 43% 0+73k 11+0io 25pf+0w
```

COMMAND FORMAT

Following is the general format of the **times** command.

```
times command
```

C Shell

```
time command
```

Arguments The following argument may be passed to the **time** command.

command Any executable UNIX command.

The output format can be controlled by setting the *time* variable. The first value assigned to the *time* value must be the amount of cpu time which must be used before the time is displayed automatically. The format tags are as follows.

Tag	*Description*
%D	Average amount of unshared data space.
%E	Elapsed (wall clock) time (mm:ss - mm=minutes, ss=seconds)
%F	Number of page faults
%I	Number of block input operations
%K	Average amount of unshared stack space used in kilobytes
%M	Maximum real memory used during executin of the process
%O	Number of block output operations
%P	Total CPU time (%U+%S) as a percentage of Elapsed time (%E)
%S	Number of CPU seconds consumed by the kernel on behalf of the user's process
%U	Number of CPU seconds consumed by the user's process outside the kernel
%W	Number of swaps
%X	Average amount of shared memory used in Kilobytes

The default format for the output of time is

```
%U %S %E %P %X+%Dk %I+%Oio %Fpf+Ww
```

To change this format you use the set command. For example

```
set time=(0 "%U user %S sys %E elapsed %P percent")
```

would display just the %U, %S, %E, and %P values after each command.

RELATED COMMANDS Refer to the **time/timex** command described in Module 136.

APPLICATIONS

The **times** command is used to see how much time you have used as a user process and as a system (kernel) process. Using **times** can keep you informed of how much CPU time you are requiring to perform your tasks.

The command is used on systems where accounting is enforced and monitored. Many UNIX systems don't enforce any type of departmental accounting and thus many users are not aware of their CPU usage. It may be beneficial for you to check your usage for comparison reasons.

TYPICAL OPERATION

In this activity you use the **times** command to display your accumulated user and system times. Begin at the shell prompt.

1. Type **times** and press **Return**. The **sed** command can be used to substitute the output of times into a more readable format. See the **sed** Typical Operation section for an example

```
cj> times
0m03.03s      3m49.08s
0m33.50s      2m43.34s
```

C Shell

```
cj> time
0.12u   0.17s   0:02   10%   0+204k   6+0io   44pf+0w
```

---

2. Turn to Module 130 to continue the learning sequence.

Module 138
touch

DESCRIPTION

The external **touch** command is used to update the access and modification times of a file. If the file given as an argument does not exist, **touch** will attempt to create it with a length of zero bytes.

BSD (Berkeley)

The BSD version of touch only updates the modification time of a file.

COMMAND FORMAT Following is the general format of the **touch** command.

```
touch [ -acm ] [ mmddhhmm[yy] ] file_list
```

BSD (Berkeley)

```
touch [ -cf ] file_list
```

Options The following list describes the options used to control how **touch** functions.

-a	Only the access time is updated. By default both access and modification times are changed.
-c	If the file does not exist do not create it. By default the file is created if it does not already exist.
-m	Only the modification time is updated.

BSD (Berkeley)

The BSD version of **touch** does not support the -a option since it does not make access time modifications. Nor does it support the -m option.

-c	Same as SV.
-f	Attempts to force a **touch** regardless of read and write permissions of the file.

Arguments The following list describes the arguments that may be passed to the **touch** command.

mmddHHMM[yy] The date and time to be used for the new access and modification times. The mm is the month (01-12), dd is the day of the month (01-31), HH is the hours (00-24), MM is the minutes (00-59), and yy is the optional last two digits of the year (00- 99). If you do not provide a year then the year is left untouched.

The current date and time are used if you do not specify them.

file_list One or more files **touch** is to update or create.

FURTHER DISCUSSION If only files are given for arguments, **touch** will use the current time to update the access and modification times of the files. If a file does not exist, it will be created with no data (the size will be zero).

If you need a file for shell script locking or date/time comparison, use the **touch** command to create it, since no disk space is used other than the name and inode. If you use **echo** or some other command, at least one block of data will be used when the file is created.

The access and modification times of a file are stored in the file's inode. The inode is the file information or header node. It is discussed in Module 2.

RELATED COMMANDS Refer to the **date** command described in Module 30 and the **ls** command described in Module 84.

RETURN CODES The return code for **touch** is equal to the number of files that could not be updated or created.

APPLICATIONS

You can use the **touch** command to update old files to prevent them from being removed by a cleanup program. The system administrator may have a cleanup program run on the system that saves and removes all files older than 90 days. If you do not want a file to be removed because of its age, you can use **touch** to update its access time.

Often **touch** is used to update program files when using the programmer's **make** utility. The **make** program compares modification times of source and object files to determine if a module needs to be recompiled. By using **touch** you can force **make** to recompile a specific file as you wish.

It may also be used to create date and time comparison files. For example, if you need to back up files that are newer than the last backup, you could type "touch .backup" and then use the "find . -newer .backup -print | cpio -oB > /dev/rmt/0m" command to save the files to magnetic tape. The next time you want to perform your backup, just use the find command and all files with a newer date than the .backup file will be saved.

TYPICAL OPERATION

In this activity you use the **touch** command to create a file with an old date and time instead of the current date and time (SV only). Begin at the shell prompt.

1. Type **touch 0101120071 myfile** and press **Return**. The file myfile is created with the date of Jan 1, 1971 and a time of 12:00.

BSD (Berkeley)

Type **touch myfile** and press **Return**. You cannot specify a date for the file or the -a option.

2. Use the **ls -l myfile** command to view the information about the new file. The following display shows the information:

```
cj> ls -l myfile
-rwxr-xr--  1  mylogin    tech        0 Jan  1 1971  myfile
```

Notice in the fifth field the length of the file is 0.

BSD (Berkeley)

The file will have the current date. Turn to the next module.

3. Type **touch -a myfile** and press **Return**. The access time is updated but the modification time remains unchanged.
4. Type **ls -lu myfile** and press **Return** to list the new access time as shown in the following display:

```
cj> ls -lu myfile
-rwxr-xr--  1  mylogin    tech        0 Jun  7 11:39  myfile
```

5. To view the modification time type **ls -l myfile** and press **Return**. Notice the date and time revealed is the original Jan 1 1971.
6. Turn to Module 53 to continue the learning sequence.

Module 139
tput (SV)

DESCRIPTION

The external **tput** command is used to initialize a terminal or query the terminfo database for terminal attribute strings. Terminal attribute strings are sequences of characters which cause the terminal to perform certain functions. They are often referred to as escape sequences. An example is the bold attribute. If your terminal can display text in bold mode, then a certain attribute string can be sent to the terminal to turn this attribute mode on and off.

COMMAND FORMAT Following is the general format of the **tput** command.

```
tput [ -Ttype ] capname [ parms ... ]
tput [ -Ttype ] init
tput [ -Ttype ] longname
tput [ -Ttype ] reset
tput [ -S <<!
capnames [ parms ... ]
!
```

Options The following options and their arguments may be used to control how **tput** functions.

-S — Reads capnames and any parameters from the standard input. Only one capname (and its parameters) is allowed per line.

-T*type* — Allows you to specify the type of terminal you are using. If you have your terminal type set correctly (TERM=vt100), then this option is not necessary. If -T is specified, the shell variables LINES and COLUMNS are not referenced. The values are read from the terminfo entry.

Arguments The following list describes the arguments that may be passed to the **tput** command.

capname — A cap name indicating an attribute used from the *terminfo(4)* database. *Capname* refers to terminal *cap*ability name from the previous /etc/termcap database. For a complete listing of cap names refer to your *Programmers Reference Manual*, Section 4, *terminfo*.

parms — If the attribute string associated with *capname* requires parameters, you must supply them as parameters to the command. The *parms* are inserted into the attribute string.

init — Causes the initialization attributes to be sent to your terminal. The initialization attributes are accessed in *terminfo* via the *is1*, *is2*, *is3*, *if*, and *iprog* capname entries. If any delays are present in the entry, they are set in the tty driver. Depending on the entry, tab expansion will be turned on or off. If tabs are not expanded, standard tabs are set every 8 spaces. If any of the necessary information is not found to perform each of these tasks, the task is ignored.

reset — Sends the reset attributes to your terminal. The *capnames* for the reset attributes are *rs1*, *rs2*, *rs3*, *rf*. If no reset attributes exist for the terminal type you specified, the initialization strings are used. The *reset* argument causes **tput** to perform the same as the *init* argument.

longname — Displays the long descriptive name of the terminal you specified. The longname is the last name on the first line of the terminal's terminfo entry.

FURTHER DISCUSSION A clever way to use the **tput** command is to use command substitution to assign the output of the **tput** command to a shell variable. Then when you need to perform the terminal function, you **echo** the variable. If you really want to get fancy, you can create a setup file with the variables being set by **tput** followed by functions that perform the **echo** command. The following examples illustrate some of the commands you might want in a setup file.

```
cj> cat term_setup
BOLD=`tput smso`        # Turn bold display on
CEOL=`tput ceol`        # Clear to the end of the line
CLS=`tput clear`        # Home the cursor and clear the screen
HOME=`tput home`        # Home the cursor
NOBOLD=`tput rmso`      # Turn bold display off
NORM=`tput sgr0`        # Set screen attributes to normal
  #
function bold   { echo "$BOLD\c" ;}
function cls    { echo "$CLS\c" ;}
function ceol   { echo "$CEOL\c" ;}
function home   { echo "$HOME\c" ;}
function nobold { echo "$NOBOLD\c" ;}
function norm   { echo "$NORM\c" ;}
```

Once you have this file created you can execute it as part of another shell by using the . (dot) command. For example,

```
. term_setup
```

would add all these variables to your local environment. If you want subprocesses to know about these variables, you can **export** them to your global environment. To use the set variables and functions you can use

```
cj> cls
```

to clear the screen. You may need to perform the bold functions in lines of text and not as functions. To do this you can use the following syntax to change the word name to bold on your output.

```
cj> echo "Type in your ${BOLD}name${NOBOLD} and press Return: \c"
```

NOTE

Some terminals do not support certain attributes and therefore **tput** will not return strings for these unsupported attributes. You will need to check your terminal's programmer's manual for its capabilities.

You may find it useful to implement a **cursor** function for shell programming. To do this you simply create the following function.

```
function cursor { tput cup $1 $2 ;}
```

To use the functions you simply type

```
cj> cursor 4 20
```

and press Return. This positions the cursor on row 4 at column 20. If your system's **tput** does not support the cup capname, you might try the following code.

```
cj> cat cursor_setup
case "$TERM" in
   ansi|vt[123]*|pc*) function cursor { echo "\033[${1};${2}" ;} ;;
                wyse*) function cursor { echo "\033a${1}R${2}C\c" ;} ;;
                    *) echo "Unsupported terminal type." ;;
esac
```

You can use the . (dot) command to add this code to your shell script. To use the cursor function, type the following,

```
. cursor_setup
cursor 4 20
```

DIAGNOSTICS AND BUGS The following error messages are displayed if a problem is encountered.

A usage message is displayed if you specify an invalid option.

-1 A -1 is displayed on the standard output if no integer *capname* attribute is defined for the terminal type specified.

unknown terminal type or no terminfo database

Displayed if the terminfo database does not exist or the terminal you specified does not exist in the terminfo database.

unknown terminfo capability capname

Displayed if the capname you requested is not valid.

RELATED COMMANDS Refer to the **stty** and **tabs** commands described in modules 126 and 130.

RELATED FILES The attribute strings associated with the *capnames* is displayed on the standard output. The **tput** command uses the following files during execution or when compiled.

/usr/lib/terminfo/?/*	Compiled terminal description database entries.
/usr/include/curses.h	Header file for compiling the tput.c source code.
/usr/include/term.h	Header file for compiling the tput.c source code.
/usr/lib/tabset/*	Files containing the appropriate escape sequences used for setting margins and tabs on more popular terminals.

RETURN CODES The following return codes have special meanings as described.

-1 The integer type *capname* is not defined for this terminal *type*.

0 If *capname* is a Boolean and the attribute is TRUE for the terminal type specified.

If *capname* is a string and the attribute associated with *capname* for the terminal type exists. The attribute string is returned to the standard output.

If *capname* is an integer a 0 is always returned. If the integer attribute for the terminal type does not exist, a -1 is displayed on the standard output.

1 If *capname* is a Boolean and the attribute is FALSE for the terminal type specified.

If *capname* is a string and the attribute associated with *capname* for the terminal type does not exist. Nothing is sent to the standard output.

If -S option is specified a return code of 1 is impossible, since there is no way to know on which line the error occurred.

2 Usage error is displayed.

3 Unknown terminal type was specified or the terminfo database does not exist.

4 The *capname* you specified does not exist in the terminfo database.

APPLICATIONS

The **tput** command is useful in controlling the terminal attributes from the shell. It provides an interface into the terminfo database so you can perform most, if not all, of the terminal control functions supported in C. You can cursor the position, clear the screen, change the screen attributes, and other various terminal functions.

As described in the Further Discussion section previously, you can use command substitution to save the output strings and then later echo them to the terminal. This provides fast screen control. The implementation of terminal attribute functions provides another level of programming ease for controlling your terminal.

TYPICAL OPERATION

In this activity you use the **tput** command to clear the screen and place the cursor in the bottom left corner of the screen. Begin at the shell prompt.

1. Type **tput clear; tput cup 24 1** and press **Return**. Notice your screen is blank and your cursor is in the bottom left corner.
2. Turn to Module 128 to continue the learning sequence.

Module 140
tr

DESCRIPTION

The external **tr** command is used to translate or delete characters. It reads from the standard input and writes to the standard output. It cannot read or write to or from files; therefore, you must use the shell's redirection symbols or pipe the input to **tr**.

COMMAND FORMAT Following is the general format of the **tr** command.

```
tr [ -cds ] [ string1 [ string2 ] ]
```

Options The following options may be used to control how **tr** functions.

-c Complements the set of characters specified in *string1*. It causes *tr* to use the characters not specified in *string1* as *string1*. For example.

```
tr -c "[a-z][A-Z][0-9]" "[^*]" < infile
```

would replace all non alphanumeric characters with a caret (^).

-d Delete the characters listed in *string1* from the input.

-s Squeezes consecutive occurrences of a character down to one character. For example,

```
tr -s < infile
```

replaces an input of "aaaaafile is a sssset" to "afile is a set."

Arguments The following arguments may be passed to the **tr** command.

string1 String of characters to be searched for on input and replaced by string2 on the output.

string2 String of characters used to replace the characters listed in string1.

The following conventions may be used in *string1* and *string2* to match ranges of characters or repeated characters.

[*c1-c2*] Represents any characters whose ASCII codes are from character *c1* to character *c2*. For example, to represent characters from a through z on the input and capital A through Z on the output, you type

```
tr "[a-z]" "[A-Z]" < infile
```

[*c1***n*] Represents *n* repetitions of character *c1*. Character *c1* may be any ASCII character code from 001 to 377 octal. If the first digit of *n* is 0, *n* is considered to be octal. If *n* is 0 or omitted, **tr** considers it "huge" and assumes that *n* is the length of string1. Thus *c1* is repeated until all characters in string1 have bee replaced by *c1*. The following examples illustrate the use of this convention.

```
tr "abcde" "[X*5]" < infile
```

changes any occurrence of letters "a," "b," "c," "d," or "e" to a letter "X."

\nnn The backslash character may be used to precede numbers that represent characters in the ASCII collating sequence. The number *nnn* may be an octal number from 001 to 377. This notation is useful when you need to represent a control character or a character not on your keyboard.

FURTHER DISCUSSION Since the **tr** command cannot read or write to or from files, the typical command resembles the following one.

```
tr "[a-z]" "[A-Z]" < infile > outfile
```

The standard input has been redirected by the shell to be the file *infile*. The standard output has also been redirected; it is sent to file *outfile*. This command converts all lowercase letters to uppercase.

The *string1* argument provides a list of cnaracters to be replaced. The *string2* argument provides the replacement characters. The first character of *string2* is used to replace the first character of *string1* and so on for all the characters in *string2*. The following example,

```
tr abc xyz < infile
```

replaces each occurrence of a with x, each b with y, and each c with z.

If the *string2* contains fewer characters than *string1*, then the remaining characters in *string1* are not affected. For example,

```
tr abcde xyz < infile
```

changes a, b, and c to x, y, and z respectively but does not change d or e.

NOTE

Older versions of **tr** used the last character of *string2* to replace all remaining characters of *string1*.

Using brackets to specify ranges of characters is a handy convention. You can specify a large set of characters using only a few characters. For example,

```
tr "[A-Z]" "[a-z]" < infile
```

translates all uppercase letters ([A-Z]) to lowercase ([a-z]).

DIAGNOSTICS AND BUGS The ASCII NULL character (\000) is deleted from the input. It cannot be placed in *string1* or *string2*.

RELATED COMMANDS Refer to the **sed** command described in Module 117.

RELATED FILES The **tr** command reads from the standard input and writes to the standard output. It cannot read or write to a file on its own. You must use the shell's redirection symbols to read and write to and from files.

APPLICATIONS

The most common use of **tr** is to convert files from all uppercase to all lowercase or vice versa. It can also be used to translate or delete strings of characters from an input stream.

TYPICAL OPERATION

In this activity you use the **tr** command to remove all punctuation from a file. Begin at the shell prompt.

1. Type **tr -d "[!-/][:-@][\[-_][{-~]" < /etc/passwd** and press **Return**. The passwd file should be displayed with only characters and numbers in it. All of the punctuation is removed.

The next activity converts the input text to one-word-per-line output.

2. Type **tr -cs "[A-Z][a-z]" "[\012*]" < /etc/passwd** and press **Return**. The -c causes all characters other than a-z and A-Z to be changed. The -s causes multiple occurrences of the new-lines to be reduced to one occurrence. The \012 is the ASCII new-line character. Notice your display is a single column of words.
3. Turn to Module 26 (SV), Module 4 (BSD) to continue the learning sequence.

Module 141
trap (sh, ksh) / onintr (csh)

DESCRIPTION

The internal **trap** command controls what action the shell performs if a certain signal (condition) is received. UNIX uses signals to communicate with processes. If the operating system experiences difficulty while running your process and decides to terminate your process, it will send a signal to the process. You can send signals to processes using the **kill** command. The **trap** command is used by the shell to catch these signals.

All signals may be trapped except for signal 9. If signal 9 is sent to your shell, it will terminate as soon as the signal is processed.

COMMAND FORMAT Following is the general format of the **trap** command.

```
trap [ action ] [ condition ]
trap [ - ] condition
trap "" condition
trap
```

C Shell

```
onintr [ - | label ]
```

Arguments The following list describes the arguments that may be passed to the **trap** command.

action A single argument to the **trap** command can be any command. When *condition* is trapped, these commands are executed. For instance,

```
trap 'tput clear; rm $TMPFILES' 1 2 3 15
```

executes the **tput** command to clear your screen. The **rm** command is also executed to remove all TMPFILES you've placed in the TMPFILES variables.

" " The signals specified are ignored. If the first argument is a NULL string (" " or ' ') the signals specified by the *n*'s are trapped but no commands are executed. For example,

```
trap "" 2
```

causes the shell to continue normal execution when you press the Delete key.

condition The number of the signal to catch. If the signal specified in condition is caught, then the commands described in the *action* part of the **trap** command are executed. Multiple signal numbers can be placed on the same **trap** command. For example,

```
trap 'rm $TMP' 0 1 2 3 15
```

executes the rm $TMP command if any one of the 0, 1, 2, 3, or 15 signals are caught.

The shell handles special conditions if specified. They are,

DEBUG	Executes the *action* after each simple command. Functions do not inherit this trap.
ERR	Executes the *action* if a command has a nonzero return code. Functions do not inherit this trap.
EXIT	Executed when the shell exits. If inside a function, the *action* is executed after the function completes, in the environment that called the function.

If no *action* is given, signal processing for each specified signal is reset to the default.

```
trap 2
```

or

```
trap - 2
```

causes the shell to abort when you press the Delete key. This is the default action if signal 2 has not been trapped.

If no *action* and *condition* are supplied, the **trap** command lists each signal being trapped and its related commands.

C Shell

-	The shell ignores all signals.
label	The shell performs a **goto** *label* when a signal is received.
	With no option the shell resets to handle signals in the default manner. That is shell scripts exit and return to the interactive login shell.

FURTHER DISCUSSION The possible signals are system dependent. It depends on your system's implementation of the UNIX signals. In AT&T UNIX System V Release 4.0 there are 22 signals. The name of each signal is usually more standard than the number; therefore, it is advisable to use the name of the signal instead of the number.

Refer to the kill command for a list of signals.

For further information about signals on your system refer to Section 2 of your Programmer's Reference Manuals.

Handling Signals You can use the **trap** command to provide a better, more controlled user proof script. By trapping certain signals you can decide whether to abort (exit) a shell script or continue processing. A few ideas you may want to consider are:

Trapping before a loop. If you want to **trap** for a signal within a loop but do not want to exit the loop, you may want to use the following **trap** statement:

```
trap "continue" 1 2 3 15
```

If you plan on performing a large number of commands if a certain signal is trapped, you may want to consider setting up functions that will perform all the necessary steps. For example, you may have a trap statement that resembles the following one or a function for each signal.

```
cj> cat ashell
trap 'rm -f $TMPFILES
tput clear
echo "<A>bort or (C)ontinue: \c"
read ANS
case $ANS in
   [cC]) continue ;;
      *) echo Goodbye!; exit 0 ;;
esac
' INT
```

The single quotes are required to pass all of the commands to the **trap** command as one argument. To clean up your code you could write a function called *sigerr2* that contains all of the code between the two single quotes in the previous trap statement. The new trap statement would be

```
cj> trap sigerr2 INT
```

RELATED COMMANDS The **kill** command described in Module 70 is used to send signals to a shell. Certain keys on your keyboard also send signals that **trap** can catch.

APPLICATIONS

You use the **trap** command to process signals in a shell script. A **trap** on certain signals allows you to perform necessary commands before exiting the shell script. The most common use is to trap for the Delete or Interrupt key being pressed and then removing temporary files before exiting the shell script.

TYPICAL OPERATION

In this activity you use the **trap** command to trap the signal your Delete key sends and display a warning message back to your terminal. Begin at the shell prompt.

C Shell

The **onintr** command is only useful in a shell script so you can write the following script to see how it works.

```
cj> cat > cignore
#!/bin/csh
onintr sigrcvd
loop:
echo -n Enter a 9 to quit or press Return
set input=$<
```

```
if ( "$input" == 9 ) then
  echo goodbye
  exit 0
endif
sleep 5
goto loop
sigrcvd:
  echo A signal was received
  goto loop
^D
```

Type **chmod 755 cignore** and press **Return**. Now test your script by running it by typing **cignore** and pressing **Return**. Now press **Ctrl-C** or **Delete** to send interrupts to the script. Type **q** and press **Return** to exit the script. Skip to step 5.

---

1. Type **trap 'echo "Oooowwww! Stop that. That hurts!"' 2** and press **Return**.
2. Now type **kill -2 $$** and press **Return**. You should receive the message *Oooowwww! Stop that. That hurts!* on your screen.
3. To reset the trap for signal 2 type **trap 2** and press **Return**.
4. To check the trap for signal 2 type **trap** and press **Return**. There should be no listing for signal INT(2).
5. Turn to Module 74 (SV), Module 112 (BSD) to continue the learning sequence.

Module 142
true

DESCRIPTION

The internal **true** command always returns an exit status of zero. It is used to guarantee a zero exit status (return code) for a conditional type statement. The shell considers a zero return code to be true and a nonzero return code to be false.

COMMAND FORMAT Following are the general formats of the **true** command.

```
true
:
```

C Shell

```
[/bin/]true
:
(1)
```

The **colon** command may also be used to return only a true exit status. In fact the Korn shell aliases the **true** command to : as shown in the following code:

```
alias true=':'
```

FURTHER DISCUSSION The **true** command lends itself well to shell programming. It is a sure-bet command that always returns a status of zero. Thus you can program infinite loops and break out based on various conditions.

The most common code using the **true** command is probably the following:

```
while true
 do
  sleep 60
  echo "Just another endless day(loop)!"
 done &
```

This code will run forever or until someone kills the process or shuts down the system. The following lines of shell code are equivalent:

```
while :
 do
  sleep 60
  echo "Just another endless day(loop)!"
 done &
```

You could also code this with the until loop using the false command.

NOTE

The **true** command returns a 0. The shell uses 0 as a true condition. Most programming languages, such as C, use 1 or nonzero numbers as true conditions. The reason for this convention is that return codes from programs are 0 if the program completed successfully and nonzero if a problem occurred.

RELATED COMMANDS Refer to the **false** command described in Module 49.

RETURN CODES The **true** command always returns a zero return code.

APPLICATIONS

The most common use of the **true** command is in looping conventions. You can use a **true** command on a while loop statement to create an infinite (endless) loop. Some other condition such as a **case** or **if** command must execute a **break** or **exit** to leave the loop. The **true** command may also be useful in testing shell scripts. You can place the **true** command on if and while loops to control the shell script and test how it performs in these conditions.

TYPICAL OPERATION

In this activity you use the **true** command to loop indefinitely while printing the time to your screen. Begin at the shell prompt.

1. Use the **vi** editor to create the following shell script. Type **vi clock** and press **Return** to enter the editor. After you have typed the shell script, exit **vi** by typing **ZZ**.

```
CLR=`tput clear`
while true
 do
  echo "${CLR}\n\n\n\n\n"
  banner "  `date +%H:%M`"
  sleep 59
 done
```

BSD (Berkeley) version

```
CLR=clear
while (1)
 do
  echo "
  "
  echo "                    "`date`
  sleep 59
done
```

This shell will display the hours and minutes (date +%H:%M) in large letters (banner) in the center of your screen. Since the first character is not a #, shell is executed as a Bourne shell script.

2. Change the modes to executable for the script by typing **chmod 777 clock** and pressing **Return**.
3. Now execute the new script by typing **clock** and pressing **Return**.
4. To stop the script press **Del** (or **Ctrl-C**).
5. Turn to Module 49 to continue the learning sequence.

Module 143
tset (BSD)

DESCRIPTION

The **tset** command sets your terminal type and stty setting when you log in to the system. It sets your erase and kill characters, sets or resets delays, and sends character sequences to initialize your terminal. This is referred to as terminal dependent initialization.

First **tset** determines the *type* of terminal being used, then performs the necessary initialization and mode settings. The type of terminal is defined in the /etc/ttytype database. Terminal *types* may be found as files under the /usr/lib/terminfo/[a-z] directories or as entries in the /etc/termcap database. If a port (tty) is not permanently wired to a specific terminal *type* (not hardwired), it will be given an appropriate generic name. For example, ports wired to modems are given a terminal type of *dialup*.

Some System V systems implement **tset**.

COMMAND FORMAT Following is the general format of the **tset** command.

```
tset [ -AIQShns ] [ - ] [ -ec ] [ -kc ] [ -m[ident][test/baud]:type ] \
     [ type ]

reset
```

Options The following list describes the options and their arguments that may be used to control how **tset** functions.

-A — Asks you for the *type* of terminal being used.

-I — Suppresses the sending of terminal initialization sequences to the terminal.

-Q — Suppresses the displaying of the "Erase set to ..." and "Kill set to" messages. Quit mode.

-S — Displays the terminal *type* that would normally be assigned to the environment variable TERM. By default **tset** generates a shell statement that assigns the terminal type to the variable TERM. The output from -S is often used when the login shell is **sh** or **ksh**. For example,

```
set -- `tset -S`
TERM=$1
```

sets TERM to the appropriate terminal *type* returned by **tset**.

-h — Forces **tset** to read the /etc/ttytype file. If -h is not specified, **tset** reads the *type* from the environment variable TERM, unless mapping (-m) is specified.

-n — Forces the use of the new BSD 4.x tty driver to be used on systems where it is available. If you are on a crt and the BAUD is greater than 1200, the CRTKILL and CRTERASE modes are set.

-s Generates the appropriate shell commands to set the environment variable TERM to the proper terminal *type*. Your SHELL variable must be set to the shell you are currently using.

Displays the terminal *type*. The terminal *type* **tset** has decided upon is written on standard output.

-e*c* Sets the erase character (see **stty**) to character *c*. The *c* defaults to Ctrl-H (Backspace). You can either type *c* directly as a Ctrl-H or use the "hat" notation. The hat notation is "^H." The caret is the ^ character (shift 6 on your keyboard). The shell default for Backspace is #.

-i*c* Sets the interrupt character (see **stty**) to character *c*. The *c* defaults to Ctrl-C. You can set it the same way you set erase.

-k*c* Sets the kill character (see **stty**) to character *c*. The *c* defaults to Ctrl-X. If *c* is not specified and the kill character has not been set, then kill is set to @ (at sign).

-m[*ident*][*test*/*baud*]:*type*

Maps a port identifier to a baud rate and terminal *type*. The -m option allows you to pass **tset** a set of conditions, which basically say "If I'm on this port then I'll probably be using this terminal *type*." Thus you "map" information into desired terminal *types*. The following list describes each argument of the -m option.

ident Port identification string. Identifies how a terminal is attached to each specific port on the system. Valid port identification strings are located in /etc/ttytype. If no identifier is given, all port identifiers are matched. So if you want your terminal *type* to always be set the same no matter what port you log in on, leave the ident field blank.

test Conditional operators used to provide a range of possible baud rates allowed for selecting the specified terminal *type*. The operators are,

Operator	*Description*
>	greater than
=	equal to
<	less than
@	equal to, at the rate of
!	invert the test condition

baud Specifies the baud rate at which the computer port must be currently set for **tset** to select the specified terminal *type*. This is retrieved via **stty**. The **tset** output to set the baud rate is the **stty** *baud* **command.**

type Same as terminal *type* listed in the arguments section.

TIP: It is advisable to place the entire argument for each -m option in single quotes.

-n Sets the tty driver to *new* mode on systems with BSD 4.X tty drivers. Other mode sets include CRTERASE and CRTKILL.

If no options or arguments are specified, **tset** reads your terminal *type* from the environment and initializes the terminal.

The following options are accepted to provide compatibility with earlier versions of **tset**. You are discouraged from using these options since they will probably disappear in future releases of **tset**.

-r Displays results to user.

-E*c* Same as -e*c*.

Arguments The following argument may be passed to the **tset** command.

type Specifies the type of terminal being used. This argument overrides the *type* specified in */etc/ttytype*. You should always use the */etc/ttytype* for hardwired terminals.

If no options or arguments are specified, **tset** reads your terminal *type* from the environment and initializes the terminal.

Setting Your Shell TERM Variable Usually, you want to set your terminal *type* when you log in to the system. You can do this by using **tset** with the -s option. In your .profile (if using the Bourne or Korn shell (**sh**)) type:

```
eval `tset -s other_options`
```

The output will be

```
TERM=vt100
```

The **eval** statement will execute the output as a shell command, thus setting your terminal type to vt100.

In your .login (if using the C shell (**csh**)) type

```
set noglob; eval `tset -s other_options`; set glob
```

The output will be

```
setenv TERM vt100
```

which is executed by the **eval** command.

Resetting Your Terminal It is not uncommon for your terminal to be left in a strange (and often useless) state by a program that aborted for one reason or another. To reset your terminal you can use the **reset** command.

Since the terminal is confused, the following command may not be displayed as you type it. First type a **Del** character (Ctrl-C or Delete key). Type <Linefeed>reset<Linefeed>. This should reset your terminal. To check the results type a UNIX command. It should display as you type it and execute after you press Return.

The **stty** modes set by **reset** are cooked, echo, and icrnl. The cbreak and raw modes are turned off. Special control characters (erase and kill) are restored to a reasonable state. If they are unknown, then they are reset to the default values.

EXAMPLE The following example shows you how to use **tset** to set the terminal type even if you log in from various ports (terminals). If you place the following command line in your .login or .profile,

```
tset -Q -e"^H" -k"^X" -m 'dialup>300:vt100' -m 'dialup:adm5' \
     -m 'tty05:?tvi925'
```

your terminal type is set as follows:

1. Suppress the verbose messages.
2. Set erase character to Ctrl-H.
3. Set kill character to Ctrl-X.
4. If the port you log in on is a *dialup* with a baud rate greater than 300, the terminal *type* is set to vt100.
5. If the port you log in on is a *dialup* with a baud rate less than or equal to 300 baud, the terminal *type* is set to adm5.
6. If the port you log in on is tty05, **tset** prompts you for confirmation of terminal type tvi925. If you press Return, tvi925 is used as the terminal type. If you are using a different type of terminal, enter the terminal *type*. This new *type* is used by **tset**.
7. If none of the conditions listed in the **tset** command are true, then the terminal *type* is set to the *type* defined in */etc/ttytype*.

RELATED COMMANDS Refer to the **ksh** and **stty** commands described in modules 71 and 126.

RELATED FILES The **tset** command retrieves information from the following files and directories.

/etc/ttytype	A database that maps port names to terminal types.
/usr/lib/terminfo/[a-z]/*	The terminal *type* names database (SV).
/etc/termcap	The terminal *type* database (BSD and some SV). Used instead of the new terminfo structure.

VARIABLES The following variables are used by **tset**.

SHELL	Used to generate the appropriate shell commands for setting the TERM variable to the correct terminal *type*.
TERM	The name of the terminal *type*.

APPLICATIONS

The **tset** command is used to initialize your terminal, set up the correct **stty** modes, and provide proper shell commands for setting the terminal *type*. It provides a flexible way of setting your terminal *type* and certain control characters to desired values.

TYPICAL OPERATION

In this activity you use the **tset** command to display the terminal type defined for the port you are currently logged in on. Begin at the shell prompt.

1. Type **tset -h** and press **Return**. The terminal type defined in /etc/ttytype for your port is displayed.
2. Type **tset -** and press **Return**. This displays your current terminal *type* as defined in the TERM variable.
3. Turn to Module 19 to continue the learning sequence.

Module 144
tty

DESCRIPTION

The external **tty** command displays the name of your terminal device path. Your terminal device path is used by UNIX to communicate with your physical terminal. The **tty** command does not display your terminal type.

COMMAND FORMAT Following is the general format of the **tty** command.

```
tty [ -ls ]
```

BSD (Berkeley)

```
tty [ -s ]
```

Options The following list describes the options that may be used to control how **tty** functions.

-l Display the terminal device path if the device is on an active synchronous line. A normal line on most UNIX systems is asynchronous.

-s Inhibit displaying any information. Used to test return codes.

FURTHER DISCUSSION Most UNIX systems allow you to access a generic terminal device path with the name /dev/tty. If this is true for your system, then you do not have to know the actual terminal device path for your terminal to perform direct input and output to your terminal.

The use of direct input and output is necessary in some cases of shell script programming. You may be redirecting the output from a while loop into a file but, within the while loop, want to redirect the output of a command to your terminal. Refer to Module 108 on Programming Constructs for further information.

DIAGNOSTICS AND BUGS

not a tty Displayed if the standard input is not a terminal and -s was not specified. The tty command was called from a shell that is executing in background.

not on an active synchronous line
Displayed if -l is specified and you are not connected via an active synchronous line.

RELATED COMMANDS The **tty** command writes to the standard output.

RETURN CODE The **tty** command returns the following return codes upon termination. They can be used to check what happened when **tty** executed. Return codes are usually used in shell scripts in conjunction with conditional **if** statements and the **test** command.

0 If standard input is a terminal, completed ok.
1 The tty command failed, various reasons.
2 An invalid option was given.

APPLICATIONS

The **tty** command can be used to display what device your terminal is connected to on the UNIX system. It is often used by the system administrator to double check which tty port you are connected to. If maintenance requires moving your terminal cable, the port must be known in most situations.

TYPICAL OPERATION

In this activity you use the tty command to find your tty path. Begin at your shell prompt.

1. Type **tty** and press **Return**. Notice a device path is returned; all UNIX device drivers reside in the /dev directory.

```
cj> tty
/dev/tty11
```

2. Type **tty -s** and press **Return**. The tty command did not return any output, but by typing **echo $?** and pressing **Return** you can display the return code of tty. The 0 (zero) reveals tty completed without errors.

```
cj> tty -s
cj> echo $?
0
```

C Shell

```
cj> tty -s
cj> echo $status
0
```

---

3. Type the following code to see how you might use the tty command in a shell script.

```
cj> if tty -s
then echo "This IS a tty!"
else echo "Sorry, you must be on a tty to run this script!"
     exit 1
fi
```

C Shell

```
cj> tty -s
cj> if ( $status == 0 ) then
echo "This IS a tty\!"
else echo "Sorry, you must be on a tty to run this script!"
endif
```

---

4. Turn to Module 66 (SV), Module 159 (BSD) to continue the learning sequence.

Module 145
typeset (ksh)

DESCRIPTION

The internal **typeset** command is used to assign attributes and values to shell variables. By assigning attributes to shell variables you can:

- Control how a variable is represented internally.
- Control how a variable is displayed.
- Set the access or scope of a variable.

The following functions can be performed using the typeset command and attributes.

- Format your shell output using the -L, -R, and -Z attributes.
- Perform rapid arithmetic using the -i attribute.
- List all variables with specific attribute set.
- List all variables set with an attribute.

COMMAND FORMAT Following is the general format of the **typeset** command.

```
typeset [ -HLRZaefilprtux [ name[=value] ] ... ]
```

Options The following list describes the options and their arguments that may be used to control how **typeset** functions.

NOTE

Notice that options preceded with a - (hyphen) turn the type or attribute on. When the option is preceded by a + (plus), the type is turned off.

-H Provides UNIX to host-name file mapping on non-UNIX systems.

-L*n* Left justify and remove leading blanks. If *n* is a positive number, set width of the variable to *n*. If *n* is not a positive number, set variable to width of the first value it is assigned. Each time the variable is assigned a value, the variable is padded with blanks on the right. If the value is too wide, it is truncated. The -R option is turned off. The -Z option causes leading zeros to be removed.

-R*n* Right justify and fill with leading blanks. If *n* is a positive number, set width of the variable to *n*. If *n* is not a positive number, set variable to width of the first value it is assigned. Each time the variable is assigned a value, the variable is padded with blanks on the left. If the value is too long, it is truncated. The -L option is turned off.

-Z*n* Right justify and fill with leading zeros, only if the -L option has not been specified. If *n* is a positive number, set width of variable to *n*. If *n* is not a positive number, set variable to width of the first value it is assigned. Each time the variable is assigned a value, the variable is zero filled if the first nonblank character is a number. If the first character is not a number, the variable is blank filled. If value is too long, it is truncated.

-e The variable is tagged as having an error. You may **set** or **unset** this attribute. The shell does not use this attribute itself.

-f Defines the variable to be a function. The only other valid options are -t, -u, and -x. You cannot assign values to a function name. The -t option turns on execution tracing. The -u option marks the function as undefined. The FPATH variable is used to locate the function when it is referenced.

-i*n* Variable is defined as an integer. If *n* is a positive number, it is used as the base for the variable. If *n* is not a positive number, the base from the first value assigned to the variable is used. By defining a variable as an integer, internal shell arithmetic is much faster (10 to 30 times).

-l Convert uppercase characters to lowercase. The uppercase option, -u, is turned off.

-p The output of **typeset** is written to the shell's two-way pipe. The pipe is established by ending a command with |&.

-r The variable is made read-only.

-t Tags the variable. You may define tags for the variable. The tags have no special meaning to the shell.

-u Convert lowercase characters to uppercase. The lowercase option, -l, is turned off.

-x The variable is **export**ed.

If only the option is supplied and no arguments are given, then all variables with the attributes of the given option are listed.

Arguments The following list describes the arguments that may be passed to the **typeset** command.

name=*value* Assigns the value to the variable referred to as name.

If no options or arguments are supplied, then all variables are listed with their attributes.

FURTHER DISCUSSION The attributes or types assigned to variables are like those used in most programming languages. The C language has types of long, integer, and char to mention a few. These types tell the computer how to interpret the data stored in the variable.

There are two attributes you can set using commands other than **typeset**. The commands are the **readonly** command and the **export** command.

The attributes of a variable are initially set by the **typeset** command. The value assigned to a variable is transformed according to the variables attributes. For example, if you assign 10 to variable NVAR that is an octal integer (typeset -i8 NVAR), the value of NVAR would be transformed to 12 (octal).

NOTE

It is important to understand and remember how the shell handles attribute assignment. When you assign a value to a variable, the value is transformed based on the attributes of the variable.

NOTE

If you change the attributes of a variable, the value may change automatically. Check your results; don't assume you know the new value.

You can perform simple substring parsing with the formatting attributes. By setting the attributes of a variable to a certain length, you can assign a value to the variable so that only part of the value is retained. For example,

```
cj> typeset -L  LSTR=abc
cj> typeset -R5 RSTR
```

will set variable LSTR to abc, a three-position, left-justified string. RSTR is set to a five-position, right-justified string. If you performed the following assignments:

```
cj> LSTR=/u1/tech/mylogin
cj> echo $LSTR
/u1
```

the result of /u1 is returned because LSTR only accepts the first three characters of a value. The following example shows how to right justify the value of a variable:

```
cj> RSTR=x
cj> echo ":$RSTR:"
:    x:
```

You can remove the attributes (and the variable) by using the **unset** command. When you use **unset** you remove the variable from the environment.

RELATED COMMANDS Refer to the **export** and **readonly** commands described in modules 47 and 112. Both of these commands assign attributes to given variables.

APPLICATIONS

You can benefit greatly from the use of the **typeset** command. You can format output to generate reports, tables, and menus. If you need to perform mathematical calculations, typesetting the variable to integer type will increase the execution speed of your shell scripts tremendously. The alphabetic case conversion provides an easy and fast method of converting between uppercase and lowercase.

The **typeset** command takes the shell programming language one step closer to a powerful high-level language, allowing you to control the data types related to variables, plus providing automatic text manipulation within variable assignment.

TYPICAL OPERATION

In this activity you use the **typeset** command to define zero filled numbers. If you are using the csh skip to the next module. Begin at the shell prompt.

1. Type **typeset -Z5 NUM** and press **Return**. This defines *NUM* as a five-place variable. The output will be 5 places, zero filled.
2. Now assign a value to NUM by typing **NUM=4** and pressing **Return**.
3. Display the NUM variable by typing **echo $NUM** and pressing **Return**. Notice that the 4 is preceded by four zeros (00004).
4. Turn to Module 84 to continue the learning sequence.

Module 146
ulimit (ksh) / limit (csh)

DESCRIPTION

The internal **ulimit** command sets an automatic resource limit. The limit applies to the current process and all processes it spawns. The limit is set as a maximum number of resources a process can have. The **ulimit** command allows you to:

- Set the maximum size of a core file.
- Set the maximum size of data segment or heap.
- Set the maximum size of a file you can create (a user can only reduce the size).
- Set the maximum file descriptor.
- Set the maximum size of stack segments.
- Set the maximum CPU time allowed.
- Set the maximum size of virtual memory.
- Display the current maximum size limit of any of the above.

C Shell

The csh version of **ulimit**, named **limit**, allows the same type of resource control as the ksh's ulimit.These limits can be removed by using the **unlimit** command.

COMMAND FORMAT Following is the general format of the **ulimit** command.

```
ulimit [ -[HS]a|cdfnstv ]
ulimit [ -[HS]c|d|f|n|s|t|v ] limit
ulimit [ limit ]
```

C Shell

```
limit [ -h ] [ resource [ limit ] ]
unlimit [ -h ] [ resource [ limit ] ]
```

Options The following options may be used to control how **ulimit** functions. If the *limit* argument is specified, then the options perform as described; otherwise, they display the current limits.

-H	Set or display hard limits. A user cannot increase hard limits.
-S	Set or display soft limits. A user can increase or decrease soft limits. The maximum soft limit must be less than the hard limit value.
-a	Display all limits.

-c	Sets the size of the core file to *limit* kilobytes.
-d	Sets the size of the data area to *limit* kilobytes.
-f	Set the size limit for files written by the current shell and its child processes to *n* blocks. The largest file you can create is *n* kilobytes.
-s	Sets the size of the stack space to *limit* kilobytes.
-t	Sets each process to a specified cpu time limit of *limit* seconds.
-v	Sets the size of virtural memory to *limit* kilobytes.
	If no option is given and *limit* is provided, the -f option is assumed.
	If no option is given and *limit* is not provided, then the current ulimit for -f (file size) is displayed.

C Shell

-h	Sets the hard limits instead of the soft limits. Only a priviledged user (root) can set hard limits.

Arguments Only one argument may be passed to the **ulimit** command.

limit	Set the file size limit to *n* blocks for all files created by the current shell and its child processes.

C Shell

Where *resource* is one of the following.

cputime	Same as ksh -t
filesize	Same as ksh -f
datasize	Same as ksh -d
stacksize	Same as ksh -s
coredumpsize	Same as ksh -c

The *limit* can be scaled in the csh. The scaling factors are:

*limit*h	Hours (cpu time only)
*limit*k	Kilobytes
*limit*m	Megabytes
mm:ss	Minutes and seconds (cpu time only)

If no arguments are provided the current limits are displayed.

APPLICATIONS

The **ulimit** command is used to control resource limits. It is beneficial in a testing environment, where you may have a program that enters an endless loop while writing to a file. By having ulimit set you can only write *n* number of blocks to disk before the system kills (terminates) your program.

It may also be used in application programs to control the size of files a user may generate while using the application.

Usually it is used to reduce the maximum file size for users. The super-user can use **ulimit** to increase the upper limit of file sizes if necessary.

TYPICAL OPERATION

In this activity you use the **ulimit** command to set your maximum file size to one. Begin at the shell prompt.

C Shell

If you are using the **csh**, replace the **ulimit** command with **limit f** in the following exercises.

Bourne and Korn Shell Only (sh and ksh)

Since the **ulimit** command resets the maximum size file you can create, you should enter a subshell. This will keep **ulimit** from changing your login shells file size limit.

To do this type **ksh** and press **Return**.

---

1. First find out what your current file size limit is by typing **ulimit** (**limit f** in **csh**) and pressing **Return**.
2. Type **ulimit 1** (**limit f 1** in **csh**) and and press **Return** to set your maximum file size to one block.
3. Now type **cat /etc/passwd > passwd** and press **Return**. Notice the cat command displays a message about the number of blocks written and an output error. The maximum size your copy of the password file can attain is one block. If your system's passwd file is less than one block in size you will not see this error.
4. List out the size of each password file by typing **ls -l /etc/passwd passwd** and pressing **Return**.

```
cj> ls -l /etc/passwd passwd
-rw-r--r--  1  root     root     8197  Jan  1 17:23 /etc/passwd
-rw-r--r--  1  mylogin  ts        512  Jan  1 17:23 passwd
```

Notice the file size in column five. The /etc/passwd file is 8197 bytes, while the new password file is only 512 bytes (1 block). The first 1 block of data was copied and the remainder was discarded.

Bourne and Korn Shell Only (sh and ksh)

Return to your login shell by typing **exit** and pressing **Return** or by pressing **Ctrl-D**.

---

5. Type **ulimit** (**limit f** in **csh**) and press **Return** to see your original ulimit setting.
6. Turn to Module 147 to continue the learning sequence.

Module 147
umask

DESCRIPTION

The internal **umask** command masks your normal file modes, basically allowing you to have a personally defined set of modes for all new files and directories you create. By using **umask** you can:

- Set the modes to be used on all new files and directories created.
- Display the current masking for modes.

The normal default for a system is 022. This masks the write permission for the group users and all other users. The result of umask 022 on a new file is permissions of 755, which is displayed by **ls -l** as:

```
-rwxr-xr-x 1  mylogin    ts          5 Jan 24 17:09 anewfile
```

COMMAND FORMAT Following is the general format of the **umask** command.

```
umask [ nnn ]
```

Arguments The following argument may be passed to the **umask** command.

nnn — The **umask** value is set to *nnn*. This value must be an octal number. It is subtracted from the standard mode of 777 column-by-column to define your new file creation mode. You may think of the umask as the modes being denied.

If you do not specify a value, the current umask value is displayed.

RELATED COMMANDS The **chmod** command is used to change modes and define the available absolute modes that can be used with **umask**. Refer to Module 17.

APPLICATIONS

The **umask** command is usually used in the system's profile file or the user's own .profile. It provides a means of controlling what modes are set for all users as defaults. You may decide you want different default modes than are being used as a system default. If this is the situation, place a **umask** command in your .profile to set your **umask** each time you log in.

TYPICAL OPERATION

In this activity you use the **umask** command to set your default creation mode to 750. Begin at the shell prompt.

1. First set your umask to all zeros by typing **umask 000** and pressing **Return**.
2. Create a file newfile1 by typing **touch newfile1** and pressing **Return**.
3. Type **ls -l newfile1** and press **Return** to list the permissions of the file.

```
cj> ls -l newfile1
-rw-rw-rw-  1 mylogin     ts       0 Jan 25 13:42 newfile1
```

Note that the permissions are set to 666 (rw-rw-rw-) even though you had your umask set to 000. Thus the system masks off permissions of 111, your system's default mask may differ.

4. Type **umask 027** and press **Return**. Now all new files you create will have the mode of 640 which is 777 - 111 = 666 - 027 = 640.
5. Type **touch newfile** and press **Return** to create a new file.
6. Type **ls -l newfile** and press **Return** to list the new file's permissions. Notice the permissions; they will look like the following display:

```
cj> ls -l newfile
-rw-r----- 1  mylogin    ts       0 Jan 24 17:20 newfile
```

7. Type **rm newfile*** and press **Return** to remove the two new files just created.
8. Turn to Module 32 to continue the learning sequence.

Module 148
uname (SV)

DESCRIPTION

The external **uname** command displays the name, node, version, release, and hardware type of the current UNIX operating system. By default only the system's name is displayed. The following section on Options describes how to display other information.

COMMAND FORMAT Following is the general format of the **uname** command.

```
uname [ -amnprsv ]
uname [ -S system_name ]
```

Options The following list describes the options that may be used to control how **uname** functions.

-a	Display all possible information. Invokes all other options except -S.
-m	Display the name of the hardware. (i.e. 68020, 386i, 3B20)
-n	Display the network nodename. The nodename is the name the system is known by on a network.
-p	Display processor type (i.e. i386, 68020, 68030, SPARC, R3000).
-r	Display the release number of the operating system.
-s	Display the name of the UNIX system. This option is the default; if no other options are listed, it is used.
-v	Display the version of the UNIX system. The version is usually set by the local system administrator.
-S	The super-user may change the system name and the nodename by using the -S option. The system name can only contain eight characters. Not supported on some systems.

RELATED FILES The **uname** command writes to the standard output.

APPLICATIONS

The most common use of **uname** is to find out which system you are using. Many users set their shell prompt (PS1) to the name of their system. This is especially useful if you have access to multiple systems. If you have your prompt set on each system to the system's name, then you seldom get lost. **uname** is often used in shell scripts where control of the script is based on the information returned by **uname**.

Another reason for the **uname** command is the UUCP facility on UNIX. The UUCP facility transfers files and executes commands remotely among UNIX systems. The **uname** command provides a way for the UUCP commands to know the name of a specific system.

TYPICAL OPERATION

In this activity you use the **uname** command to display all of the information about your system. Begin at the shell prompt.

1. Type **uname -a** and press **Return** to display the information about your system.

   ```
   cj> uname -a
   cj   cj  4.0  1  80386
   ```

   The first cj is the system name, the second is the nodename, 4.0 is the UNIX release, 1 is the version, and 80386 is the hardware type.

2. Type **PS1="$"** and press **Return**. This resets your shell prompt to the system default.

3. Type **PS1="`uname`> "** and press **Return**. This sets your shell prompt to the name of your system.

   ```
   $ PS="`uname`> "
   cj> _
   ```

4. Turn to Module 85 to continue the learning sequence.

Module 149
uniq

DESCRIPTION

The external **uniq** command reports duplicate lines in a sorted file. It reads in lines and compares the previous line to the current line. Depending on the options specified on the command line it may display only unique lines or one occurrence of repeated lines or both types of lines. The default output is to display lines that only appear once and one copy of lines that appear more than once.

COMMAND FORMAT Following is the general format of the **uniq** command.

```
uniq [ -cdu [ +n ] [ -n ] [ input [ output ] ]
```

Options The following options are used to control how **uniq** functions.

-c	Precedes each line with a count of the number of times it occurred in the input.
-d	Deletes duplicate copies. Only one line out of a set of repeated lines is displayed.
-u	Displays only lines not duplicated (uniq lines).
-n	Ignores the first *n* fields of an input line when comparing for duplicate lines. A field is a string of nonblank characters. A blank is a space or tab.
+n	Ignores the first *n* characters of an input line when comparing for duplicate lines.

Arguments The following arguments may be passed to the **uniq** command.

input	The name of the file containing the input data.
output	The name of the file to hold the output data. If no output file is specified, the output is displayed on the standard output.
	The input and output files must not be the same name. If they are, the contents of the file are destroyed.
	If no input file is specified, **uniq** reads from the standard input and writes to the standard output.
	You cannot specify an output file without specifying an input file.

RELATED COMMANDS Refer to the **sort** command described in Module 122.

RELATED FILES The **uniq** command can read from the standard input or from an input file. It writes to the standard output or to an output file.

APPLICATIONS

You can use **uniq** to reduce duplicate lines from a file. First the file must be sorted, then you can remove the duplicate lines, reducing the size of the file.

It is also useful to filter out multiple blank lines from unsorted or sorted output of other commands. For example, the **dircmp** command displays its output using **pr**; thus the output usually scrolls off your screen before you can read it. But if you pipe the output of the **dircmp** command through the **uniq** command, the blank lines are reduced and the output is compact.

TYPICAL OPERATION

In this activity you use the **uniq** command to display a unique set of lines from a file that has already been sorted. Begin at the shell prompt.

1. Type **vi unsorted** and press **Return**. Enter the following lines of text, pressing **Return** after each line.

```
cj> vi unsorted
ia
b
c
c
b


a
z
y
x<Esc>ZZ
```

2. Type **uniq unsorted** and press **Return**. Notice the only change to the data is that three blank lines are reduced to one line.
3. Type **sort unsorted | uniq** and press **Return**. The output should be sorted alphabetically and a reduced listing compared to the contents of the file.
4. Type **rm unsorted** and press **Return** to remove the *unsorted* file.
5. Turn to Module 123 to continue the learning sequence.

Module 150
users (BSD)

DESCRIPTION

The external **users** command displays a list of users currently logged in to the system. The output is compacted onto a single line.

COMMAND FORMAT

The format of the **users** command follows.

```
users
```

RELATED COMMANDS

Refer to the **who** command described in Module 159.

RELATED FILES

The **users** command reads user information from the /etc/utmp file.

APPLICATIONS

The **users** command is useful when you just want to find out which users are logged in at the current time.

TYPICAL OPERATION

In this activity you use the **users** command to list all current users cn the system. Begin at the shell prompt.

1. Type **users** and press **Return**. You see a single line of users listed on your terminal. The output line may wrap onto the next line if there are many users logged on the system.
2. Turn to Module 61 to continue the learning sequence.

Module 151
vi/view

DESCRIPTION

The **vi** editor is a powerful interactive full-screen editor built on top of the **ex** line editor. Thus it provides the user with a powerful line editor for substitutions and interfacing with UNIX and a powerful screen editor for visual editing. The vi editor acts as a window into a file being edited and reflects changes as they are made.

There are various methods of editing files using **vi**. They are:

- Single file editing.
- Edit multiple files without leaving the editor.
- Specify **ex** commands to be interpreted and executed before full screen.
- Recover files lost because of abnormal termination.
- Using the tags file to edit multiple files based on tag names.
- Edit a file in read-only mode; **view** a file.
- Enter **vi** from the **ex** editor.

COMMAND FORMAT Following is the general format of the **vi** command.

```
vi     [ -CllRx ] [ -rlost ] [ -ttag ] [ -wn ] [ +cmd ] [ file_list ]

view   [ -CllRx ] [ -rlost ] [ -ttag ] [ -wn ] [ +cmd ] [ file_list ]

vedit  [ -CllRx ] [ -rlost ] [ -ttag ] [ -wn ] [ +cmd ] [ file_list ]
```

BSD (Berkeley)

```
vi     [ -l ] [ -rlost ] [ -ttag ] [ -wn ] [ +cmd ] [ file_list ]
```

---

vi Invokes **vi** in full screen mode with default settings.

view Invokes **vi** in read-only mode, no updates allowed.

vedit Designed for beginners. The report flag is set to 1, so all changes are reported. The showmode and novice options are set for more interactive information.

Options The following list describes the options and their arguments that may be used to control how **vi** functions.

-C Encryption, the data is assumed to be encrypted. You are prompted for an encryption/decryption key.

-l Sets the showmatch and lisp mode options. The (), {}, [[and]] commands are changed to indent for lisp code.

-L	List all preserved files. Files saved as a result of an editor abort or a system crash.
-r [*lost*]	Recover *lost* file that was preserved during abnormal termination of **vi**. Use -r option alone for list of preserved files.
-R	Read-only mode, no updates allowed.
-t *tag*	Starting at *tag*, edit the file that is referenced by *tag* in the "tags" file.
-w*n*	Set default window size to *n* lines.
-x	Edit encrypted files. You are prompted for an encryption key.
+*cmd*	Display buffer after interpreting an **ex** command *cmd*. Plus (+) with no *cmd* begins editing at last line of buffer. Enclose commands with special characters in single quotes ('*cmd*') to escape **ksh** interpretation.

Arguments The following argument may be passed to the **vi** command.

file_list	One or more files to edit. If you specify more than one file, **vi** edits the first file on the command line. To edit the next file you enter :n and press Return.

Table of Contents

FUNCTIONAL OVERVIEW 592
- Interface capabilities 592
- Moving around the buffer 592
- Manipulating Text 592
- The Design of vi 592
- Editing with vi 593
  - Pros 593
  - Cons 593

GETTING STARTED 593
- Determining Your Terminal Type 593
- Defining the Terminal Type 593
- Defining the Terminal Capability Database 594
- Automatically Defining TERM 594
- Emulating Terminals and Software 594

FILE RECOVERY 594
- Killing a vi Session 595
- Delete/Interrupt Key 595

THE DISPLAY 595

MODES 596

DEFINITIONS 596
- Units (Objects) 597
- Regular Expressions 597
- Registers 597
- Buffers 598
- Unused Characters 598
- Limits 599

MOTION COMMANDS 599
vi 599
Character Movement 599
Word Movement 600
Line Movement 600
Vertical Movement 600
Horizontal Movement 600
Sentence 600
Paragraph 600
Section 600
Screen 600
Character Search by Line 601
String Search 601
Marking and Returning 602
Buffer Movement 602
Scrolling 602
Go to Location 602
Screen Adjusting 602
Scrolling 603
Ex Mode 603
Miscellaneous 603

MANIPULATION COMMANDS 603
Inserting Text 604
Special Insert Command Characters 604
Operators 604
Changing Text 605
Deleting Text 605
Replacing Text 605
Indenting Text 605
Filtering Text 605
Moving Text (cut & paste) 606
Yanking (cutting) 606
Putting (pasting) 606
Moving text between files 606

MACROS 607
String Macros (Mapping) 607
Command Mode String Macros 607
Insert Mode String Macros 607
Input Mode motion definitions 608
Register Macros 608

ABBREVIATIONS 608

TRANSPOSING TEXT 609

UNDO, REDO, and RETRIEVE 609

MISCELLANEOUS 609

LEAVING vi 609

SUBSTITUTIONS 610
TAGS 610
vi QUICK REFERENCE 610
CUSTOMIZING OPTIONS 610

FUNCTIONAL OVERVIEW Once you have entered **vi** the cursor is like a pencil and eraser that may be moved around on the screen to write (insert) or erase (delete/change) as needed. The buffer can be thought of as a scroll of paper with a picture frame placed over the flat exposed part of the scroll. The data inside the frame can be visually manipulated using **vi**. The data on the entire scroll can be globally manipulated using **ex**, the line editor.

Interface capabilities The **vi** editor can interface with **ex** commands to perform global substitutions and file utilities. A subshell may be entered from **vi** and UNIX commands executed until a Ctrl-D or exit is entered. UNIX commands can be executed from **vi** without leaving the editor. UNIX filters can be interfaced to allow for special editing capabilities.

Moving around the buffer There are motion commands to move the cursor one of four directions (i.e. up, down, left, right) in multiples of units. Units are defined as:

characters	words
lines	sentences
paragraphs	sections
the screen	pattern searches
marked addresses	the entire buffer

For example, to move forward a word you press w; to move to the top of the screen you press H. There are key sequences that allow you to perform various motion commands.

Manipulating Text There are operators which may be combined with a motion command to perform the following text manipulations:

Operator	*Description*
c	change
d	delete
<	remove indent
>	indent
y	yank (same as cut)
!	filter (send text to UNIX filter and retrieve output)

For example, to change a word you would type cw; to delete everything from the current line to the top of the screen you would type dH. The c in cw is used for changing text and the d in dH is used for deleting text.

A filter is a UNIX command that can read from the standard input and write to the standard output; therefore, it can be used in a pipe. A pipe is a combination of UNIX commands with the output of one command being used as the input of the next command. Any UNIX filter command may be used by vi to manipulate the text stored in the session buffer, thus vi has all the text manipulation features of UNIX itself.

The Design of vi There are a few differences between **vi** and PC type editors or word processors that are worth noting. When comparing and using **vi**, it helps to remember that it was designed by

programmers for programmers who were using terminals and keyboards with limited capabilities. The **vi** editor was designed before fancy keyboards were available and thus its keystrokes are usually related to the functions they perform. For example, pressing i places you in insert mode.

Editing with vi The structure of **vi** is based on units of text and operators. The main focus behind vi is single keystroke commands to move around the file and the use of operators to allow text manipulations based on these commands. Any operator may be used to manipulate any unit of text. Thus vi provides a rapid way of making changes but is not user friendly.

Pros

- Motion commands and text operators can be combined.
- Interfaces with any UNIX filter command.
- Provides a way to change a unit of text, instead of inserting new text and deleting the old text, when overlaying will not suffice.
- 26 registers to store data in and retrieve at a later time.
- Multiple methods of entering Insert mode; based on units of text.

Cons

- Must toggle between modes (Command/Insert) to move or make changes.
- No true overlaying of text mode. You must be in Command mode to move around the buffer.
- Lack of inverse video for certain editing functions. Addresses marked in text are not visible.

The vi editor was developed by The University of California at Berkeley, Computer Science Division, Department of Electrical Engineering and Computer Science.

GETTING STARTED There are a few shell environment variables that must be set before you start using **vi**.

Determining Your Terminal Type List the terminfo directory for terminal types beginning with the first letter of your terminal's model number. For example, if you have a terminal with a model name of tvi955 then you would type the following command to list all possible entries.

```
$ ls -C /usr/lib/terminfo/t/tvi*
```

You now have to choose one of the listed terminal types. For further information check with your system administrator and read the terminfo entry in section 4 of the Programmers Reference Manual distributed with your system.

Defining the Terminal Type **vi** must know what type of terminal you are using so it can control your terminal screen correctly. The shell variable TERM is used by **vi** to know what terminal type you have. Therefore, you must set and export the TERM variable before invoking **vi**. For instance, the following command sets and exports TERM to be a vt100 type terminal.

```
$ export TERM=vt100
```

The model number or name label on the front or your terminal is usually used in the database if a definition exists.

CAUTION

If you do not define your terminal type correctly, **vi** may send the incorrect screen control sequences and your terminal screen may become garbled. If this happens type capital Q followed by set term=*type* and press Return, where *type* is the correct terminal type. Then type vi and press Return to reenter vi full screen mode.

Defining the Terminal Capability Database **vi** must also know where to find the control codes used to control your terminal. These codes are stored in a database referred to as TERMINFO. The TERMINFO database is located in /usr/lib/terminfo/?/*. Where the ? is the first character of your terminal type and the * represents all terminal type definitions starting with letter ?. If you wish to use your own definition of TERMINFO you can define the TERMINFO variable to the directory containing the entry. For example,

```
$ export TERMINFO=/u1/tech/mylogin
```

Csh Shell

```
setenv TERMINFO /u1/tech/mylogin
```

---

sets the TERMINFO variable so **vi** can use your copy of the terminal capability definitions instead of the system default definitions.

Automatically Defining TERM You should define your TERM variable in your .profile file. If you login from different terminals, you may want to prompt for the TERM type. The following code prompts for the TERM type and sets it accordingly.

```
TERM=vt100
echo "TERM=$TERM:\c"
read xterm
TERM=${xterm:=$TERM}
export TERM
```

If you press Return, vt100 is used; otherwise, the new type you entered is assigned to TERM. If you always use the same terminal to log in to the system, then you may want to use the code that sets your terminal type and remove the prompting code.

Emulating Terminals and Software Certain terminals have the capability to emulate other terminals. If your terminal is not a valid type, check your owner's manual for possible terminal types your terminal can emulate.

Terminal emulating software (e.g. PC communication packages) may be used to interface with vi. Certain software packages use the Esc key for their attention interrupt. You will need to change the definition of this interrupt to one of the keys not used by vi (refer to the section on Unused Characters) or to an <ALT> key on your PC.

FILE RECOVERY If you are in vi and for some reason you get logged off the system, use the recovery feature. Possible reasons are:

- If the line to your terminal dropped.
- If **vi** hung (will not respond) and you killed the session.
- The system crashed gracefully (don't hold your breath!).
- The system was shutdown by the super-user.

To recover a lost file you must first know whether **vi** "preserved" the file. The command to do this is:

```
vi -r
```

A list of all lost files is displayed. If no files are displayed then you have no files available for recovery. If a file you need to recover is displayed, you must first change your present working

directory to be the same as where the file is located. Then you can recover the file. For example, if you have a file, myprog.c, in your HOME/bin directory, to recover it you need to enter the following commands,

```
cd $HOME/bin
vi -r myprog.c
```

The file you are recovering must not have been modified since the date of preserve file. If you recover from an old preserve file, you can cause serious damage to your existing file.

Killing a vi Session You may have to terminate a session of **vi** using the **kill** command. If **vi** hung or you where disconnected from the system for some reason and **vi** did not die from the hang-up, you will have to kill the session. To do this you must first find the PID of the **vi** process. The following code illustrates the necessary commands to find the PID and kill **vi**.

```
$ ps -u mylogin | grep vi | nawk '{print $1}'
1506
$ kill -15 1506
$ vi -r oldfilename
```

The **ps** command returns a PID number if there is a hung **vi** process, in this case PID 1506. The PID 1506 is a hung **vi** session. If there is no **vi** process hung, only your prompt will return. If a PID is returned, terminate it via the **kill -15 1506** command. This causes **vi** to generate a preserve file for recovery. If no modifications were pending to be saved, then **vi** exits without preserving. If you had made changes, recover your file using the -r option.

NOTE

If the system crashed, you might try to recover your lost file by using the -r option. This will work if **vi** had enough time to preserve your **vi** session to disk.

Delete/Interrupt Key Press the Delete key to abort commands in **vi**. This may garble the screen, so be ready to use the Ctrl-R or Ctrl-L command to redraw the screen.

NOTE

If you press Delete while making a change (manipulation), you may need to use the u command (undo) to return to the condition before you began the change.

THE DISPLAY Certain characters and lines are used to present a more understandable screen picture of the session buffer.

Displayed	*Represents*
@	Deleted lines; only used on dumb terminals or if last line of display is too long and wraps off the screen.
~	Lines past end-of-file; an empty file will have a single column of tildes from the cursor to the bottom of the screen.
^x	Control characters (x = A-Z, ? or]); control-I (tab) and control-J (new-line) are always interpreted by **vi** and cannot be escaped.
Lastline	Used for general information, **ex** mode line, error messages, feedback about i/o, report notices, and to echo back input for search commands.

Command line for **vi** to create a new file.

```
cj> vi text
```

Initial screen when **vi** is invoked for a new file:

```
~
~
~
~
~
~
~
~
"text" [New file]
```

The default number of lines used for the screen display depends on the baud rate of your terminal. The faster the baud rate, the larger the initial number of lines displayed. The lines beginning with a tilde (~) are not in the session buffer. This file currently has no lines in the session buffer. The last line displays information about the current file.

Since there is no text in the buffer, you cannot move the cursor. Therefore, you must press one of the insert keys and insert text.

MODES The **vi** editor has three modes of operation. Each mode is a separate entity from the other two. You must be in command mode to issue commands to **vi** and to access the **ex** mode command line. You must be in insert mode to insert text. Thus you have to toggle between insert and command mode. The reason for this environment is simple: **vi** lets you use single keystroke commands. You do not have to twist your hand around to press some special control sequence to move the cursor around the buffer. Indeed there are some control characters used, some in command mode and a few in insert mode.

vi Mode	*Description*
Command	Interprets keystrokes and performs specified functions. The movement of the cursor by units within the buffer. The deleting, filtering, and moving of text is done in command mode.
Insert	The changing, inserting, and replacing of text by units within the buffer.
Ex/Last_Line	The movement, changing, replacing, moving, and deleting of text in conventional line editor commands within the buffer. File handling commands, status information, and substitutions are performed in **ex** mode.

DEFINITIONS Before continuing the discussion of **vi** it is necessary to understand the terminology relating to the editor. The following sections clarify some of the definitions and concepts of **vi**.

Units (Objects) Units (often referred to as objects) of text are predefined amounts of text used by motion commands.

Units	*Description*
character	Any single ASCII character (000-177 octal).
word	Text delimited with space, tab, or new-line characters.
line	Physical (UNIX) line ending with new-line or Return.
sentence	Text ending with a ., !, or ? followed by two spaces or a blank line.
paragraph	A Memorandum Macro (mm) or Typesetting (troff) token; .IP, .LP, .PP, .QP, .P, .bp or a blank line.
section	A Memorandum Macro (mm) or Typesetting (troff) token; .NH, .SH, .H, {, blank line, form feed or new-line.
screen	The visually displayed data, from the top left corner to the bottom right corner.
search	The unit of text from the current position up to but not including the next occurrence of a given search pattern.
mark	From the current position to the position marked by single letter address.
buffer	A copy of the entire buffer, from line 1 to the end of the buffer (file).

NOTE

A character preceded by a caret (^) that cannot have the cursor positioned under the ^ is a control character placed in the buffer.

Regular Expressions Regular Expressions are patterns used to match strings in searches and substitutions. Refer to Module 113 on Regular Expressions.

Registers The **vi** editor maintains various registers where text may be stored. Registers are used to save and retrieve text. You reference named registers with a single alphanumeric name. To access a register you place a double quote (") before the desired register name. For example, if you want to store a line of text in register "a" for later use, you type the following command,

```
"ayy
```

This informs **vi** to place the next *unit* of text in the "a" register. The unit of text is the current line (yy, yank one line). Possible register names are unnamed/undo, numbered, and named.

The following table describes all of the registers and buffers used by **vi**.

Name	*Type*	*Description*
0	Unnamed Register	Where the last text that was manipulated is stored. The *undo* register. You cannot address this register with the " notation.
1-9	Number Register	Stores whole lines of text deleted from the *session* buffer. As new yanks or deletes are performed, the text in registers one through nine is shifted up to the next higher register, thus the data in register nine is lost but replaced by the data in register eight.

a-z	Named Register	Save to register whose name is a single alpha character for later use. Used to move text within and between buffers (files).
A-Z	Named Register	Append to register for later use; single capital alpha character name.
REG	Register	Refers to any of the above registers.
"*REG*	Register	Address the specified register. Use the double quote to address any of the above registers, except the undo register.

Buffers The **vi** editor uses a session buffer to maintain a copy of the file you are editing. There are two session buffers that you can switch between. For example, if you enter **vi** you can edit another file without leaving **vi** and then switch back and forth between the two session buffers. The following commands illustrate how you perform this simple task:

```
vi file1        # edit the first file
:e file2        # while in vi edit the second file
:e #            # toggle back to edit the first file
:e #            # toggle back to edit the second file
```

There are two session buffers. The current session and the alternate session. You reference the alternate session buffer by using the :e# command.

The following table describes the buffers used by **vi**.

Buffer Name	*Description*
session	The editing buffer **vi** uses to store the current file being edited. **vi** does not use the name *session*; when you enter **vi** you're in an editing *session* and thus the buffer is called the current *session* buffer.
%	Name of *current* session buffer or filename.
#	Name of *alternate* session buffer. The last filename referenced in **ex** mode will become the *alternate* file.

NOTE

Each of the above buffers are dynamic ranging in size. The undo and numbered buffers are only affected by line changes and cannot be addressed for text storage.

Unused Characters These characters may be used for the name of a string macro without interfering with existing commands. String macros are created using the :map command.

Unused Command Mode Characters

Ctrl-A	Ctrl-C	Ctrl-I	Ctrl-K
Ctrl-O	Ctrl-Q	Ctrl-S	Ctrl-T
Ctrl-V	Ctrl-W	Ctrl-X	Ctrl-Z
Ctrl-\	Ctrl-_	*	\
K	V	g	q
v			

NOTE

Ctrl-Q is used for restarting output to terminal. Ctrl-S is used for stopping output to terminal; see Module 126 on **stty**. Systems with job control use the Ctrl-Z to suspend jobs.

Limits The **vi** editor has some limits that you may need to know. These limits control how **vi** functions. They may vary from manufacturer to manufacturer.

The following table lists the maximum allowed limits.

Maximum	*Description*
512 *or* 1024	Characters on one line.
250,000	Lines per file (buffer).
128	Characters per filename.
30	Characters in a tag name.
256	Characters in a global command list.
128	Characters in previous insert or delete in **vi**.
100	Characters in UNIX escape command.
32	Total string macros (map macros).
512	Total characters in string macros.

NOTE

The number of tag file entries appear unlimited. However, if you use single-digit tags the first ten tags are often misinterpreted or not understood at all.

MOTION COMMANDS *Motion Commands* allow you to move around the file in *units* of text. You must be in *Command* mode to use the motion commands. Refer to the section on Units for definitions of text. You can return to *Command* mode from any other mode by pressing Esc.

vi is in *Command* mode after invocation. **vi** interprets keystrokes and then performs the action. Some of the motion commands may be preceded by a number to repeat the command a number of times. In the following descriptions the *n* signifies that multiple repetitions of the command may be specified. In most cases *n* defaults to 1 if you do not specify a number. Not all versions support multiplicity on all commands.

Line 1 of the file is considered the top of the buffer; therefore, backward, up, and previous refer to moving towards the top of the buffer (line 1). Likewise, forward, down, and next refer to moving towards the bottom of the buffer (last line). If the bell "beeps," then you are at a buffer boundary or the command is not valid.

Character Movement There are four commands to move forward and backward by a character unit.

*n*h
n<bs> — Move backward *n* characters.

*n*l
n<sp> — Move forward *n* characters.

To move right one character press an l or Right Arrow; to move left one character press an h or Left Arrow.

Word Movement There are six commands used for moving by words. A word is defined as any set of characters delimited by a tab, space, or new-line.

*n*w	Move forward to the beginning of the *n*th word.
*n*W	Move forward to the beginning of the *n*th word, ignoring punctuation.
*n*e	Move forward to the end of the *n*th word.
*n*E	Move forward to the end of the *n*th word, ignoring punctuation.
*n*b	Move backward to the beginning of the *n*th previous word.
*n*B	Move backward to the beginning of the *n*th previous word, ignoring punctuation.

Line Movement There are eight line-oriented commands. A line is defined as beginning at the left side of the screen and ending with a carriage return (octal 15). A physical line may wrap around the screen and create what appears to be multiple lines.

Vertical Movement allows you to move from line to line.

*n*j *n*Ctrl-N	Move forward (down) *n* lines and keep the cursor in the same column.
*n*k *n*Ctrl-P	Move backward (up) *n* lines and keep the cursor in the same column.
n+ *n*<cr>	Move to beginning of text on the next line.
n-	Move to beginning of text on the previous line.

Horizontal Movement allows you to change positions on the current line.

0	Move to beginning of current line (zero).
n$	Move to end of current line.
^	Move to beginning of text on the current line.
n|	Move to column *n*. If *n* is not given then move to beginning of current line.

Sentence There are two commands for moving by sentences. A sentence refers to the English grammar version of a sentence ending with punctuation.

n)	Move to beginning of next sentence.
n(	Move to beginning of previous sentence.

Paragraph There are two commands for moving by paragraphs. A paragraph ends with a Memorandum Macro Paragraph token or a blank line.

n}	Move to beginning of next paragraph.
n{	Move to beginning of previous paragraph.

Section There are two commands for moving by section. A section ends with a Memorandum Macro Header token (.H) or an open or close brace ({}) in C program text.

n]]	Move to beginning of next section or function.
n[[	Move to beginning of previous section or function.

Screen There are three commands that allow you to move around based on the displayed screen.

*n*H	High point (home; top left corner) of screen. Move cursor to the beginning of first (top) line displayed on the screen.

M	Middle point (middle line) of screen. Move cursor to beginning of middle line displayed on the screen.
*n*L	Low point (bottom left corner) of screen. Move cursor to beginning of last line displayed on the screen.

The following examples illustrate the various screen movement commands.

5H	Locate the cursor 5 lines below the top of the screen.
L	Locate the cursor on the last line of the screen.

Character Search by Line You can use one of the following six commands to search for a particular character on the current line. If no character is found, then **vi** displays a video bell or beeps at you.

nfc	Search forward on current line to character *c* in text.
*n*F*c*	Search backward on current line to character *c* in text.
ntc	Search forward on current line to position before character *c*.
*n*T*c*	Search backward on current line to position after character *c*.
n;	Repeat last *f, F, t,* or *T*.
n,	Inverse direction of last ; command.

The following examples help clarify the use of the character search commands.

ft	Search for the first occurrence of t on the current line.
Fx	Search backward for the previous occurrence of the letter x.
;	Search backward for the next occurrence of x.

String Search String searches are used to locate an occurrence of a specific string. Regular expressions may be used to help locate various different strings using the same search pattern. When you type the slash (/) your cursor jumps to the bottom left corner of your screen. At this point enter the pattern you wish to search for in the buffer.

n/pat	Search forward for the string matching *pat*.
n?pat	Search backward for the string matching *pat*.
n/pat/+x	Search forward for the string matching *pat*, then move forward (down) *x* lines.
n?pat?+x	Search backward for the string matching *pat*, then move forward (down) *x* lines.
n/pat/-x	Search forward for the string matching *pat*, then move backward (up) *x* lines.
n?pat?-x	Search backward for the string matching *pat*, then move backward (up) *x* lines.
*n*n	Repeat last / or *?*, *n* times in same direction.
*n*N	Reverse the direction of last / or *?*, repeat *n* times.
n//	Search for *n*th match of last pattern.
;	Search command separator.
/pat1/;/pat2/	Search for *pat2* after finding *pat*.

The following examples illustrate the use of the search command. End each search pattern by pressing Return. If a pattern match is located, the cursor is positioned at the new location. If no string is located, **vi** notifies you as such.

/Myname	Searches for the literal string "Myname."
/[A-Z0-9]	Searches for an uppercase letter or a number.
/^[^Ctrl-I]	Searches for a line that does not begin with a tab (Ctrl-I).
/^$	Search for blank (empty) lines.

Marking and Returning

''	Move to prior location in buffer, beginning of line. Functional only after text manipulation or a motion command is performed.
``	Move to prior location in buffer, exact character position on line. Functional only after text manipulation or a motion command is performed.
m*c*	Mark current cursor position with character *c*. Valid characters for *c* are *a-z*. The mark is invisible, no character or reverse video features are provided.
'*c*	Move to the beginning of the line marked by character *c* (see m*c*). Includes last line of defined text when used with an operator.
`*c*	Move to the exact position marked by character *c* (see m*x*). Excludes last line of defined text when used with an operator.

Buffer Movement These following six commands are used to scroll the buffer forward or backward to display a new screen or to jump to a specific line and display a new screen.

Scrolling

*n*Ctrl-F	Scroll *f*orward (down) *n* full screens.
*n*Ctrl-B	Scroll *b*ackward (up) *n* full screens.
*n*Ctrl-D	Scroll forward (*d*own) *n* half screens.
*n*Ctrl-U	Scroll backward (*u*p) *n* half screens.

NOTE

The Ctrl-F, Ctrl-B, Ctrl-D, and Ctrl-U commands are not true *motion* commands and *cannot* be used in conjunction with an *operator*.

Go to Location There are two ways to move to specific locations in the buffer. One is by line number, the other is by symbolic tags.

n*G*	*G*o to line *n*. If *n* is not given then cursor is moved to last line of the buffer (file).
^]	Go to tag described in next word, refer to section on Tags.

Screen Adjusting

*n*Ctrl-E	Scroll window forward (down) 1 line.
*n*Ctrl-Y	Scroll window backward (up) 1 line.
Ctrl-L	Redraw screen (cleans garbled screen).
Ctrl-R	Redraw the screen removing all lines with @ signs. Dumb terminals cannot remove deleted lines from the display and thus **vi** places an @ sign on the line to signify that it has been deleted. Ctrl-R redraws the screen, removing all of these deleted lines.
z	Redraw, place the current line at top of the window.
z.	Redraw, place the current line at the middle of the window.
z-	Redraw, place the current line at the bottom of the window.
*n/pat/*z	Redraw, place the line with *pat* at top of window.
*n/pat/*z.	Redraw, place the line with *pat* at middle of window.
*n/pat/*z-	Redraw, place the line with *pat* at bottom of window.
zn.	Redraw, use *n* line window.

NOTE

"Window" refers to the current window size, which may or may not be the entire physical screen. The window size expands until the screen size is reached as the cursor is moved outside the window limits. Different terminal types cause different actions to occur when screen manipulation is performed.

Scrolling You can use the following six commands to scroll the buffer forward or backward to view text a screen at a time.

*n*Ctrl-E	Scroll the display window forward (down) *n* lines.
*n*Ctrl-Y	Scroll the display window backward (up) *n* lines.
*n*Ctrl-F	Scroll *f*orward (down) *n* full screens.
*n*Ctrl-B	Scroll *b*ackward (up) *n* full screens.
*n*Ctrl-D	Scroll forward (*d*own) *n* half screens.
*n*Ctrl-U	Scroll backward (*u*p) *n* half screens.

Ex Mode The following commands may be used to interface with the **ex** line editor and return to the **vi** screen editor.

Q *q*	Enter ex mode, return to **vi** mode with **:vi**. The **ex** mode is the same as entering and using the **ex** line editor. You may want to enter **ex** mode if you have a lot of substitutions to perform.
:	Enter temporary **ex** mode. Precede each **ex** command you need to perform with a : (colon); this causes **vi** to send the command to **ex** for processing.
:vi	Enter/Return to **vi** mode. If you are in **ex** you can enter or return to **vi**.
:open	Enter to *open* mode.

Miscellaneous

&	Repeat last substitute command.
%	Find matching ( ), [ ], or { }.
*n*G	*G*o to line *n* (default: last line).
Ctrl-G	Give status of current file.
Ctrl-]	Go to tag described in next word.
Ctrl-^	Edit alternate file if two files are being edited and no updates since last write (see *:e #*).
Q	Enter **ex** mode.
q	
:	Enter temporary **ex** mode.

MANIPULATION COMMANDS *Text Manipulation* is not a true **vi** mode. It is a combination of the text manipulating commands of *command* mode and all the commands of *insert* mode. To leave *insert* mode and return to *command* mode, you MUST press Esc. If at any time you feel lost or wish to abort a command, press Delete or Esc. The *i*nsert, *a*ppend and *o*pen commands all enter the *insert* mode. A dollar sign ($) will appear at the end of *units* of text that are being changed.

NOTE

Deleted/Yanked text is stored in the *undo* register unless a register is defined, refer to the section on Registers and Buffers.

Inserting Text These commands allow various ways to insert text into the file. You must press Esc or Delete to exit insert mode. The text (txt) in the following commands may be any amount of text you need to type. It is advisable to press Return after each sentence punctuation and have your wrap margin set to a reasonable number. One sentence should be broken into multiple lines to make later editing easier.

*a*txt<Esc>	*A*ppend new text after the cursor on the current line.
*A*txt<Esc>	*A*ppend new text at the end of the current line.
*i*txt<Esc>	*I*nsert new text before the cursor on the current line.
*I*txt<Esc>	*I*nsert new text before the first text on current line.
*o*txt<Esc>	*O*pen the buffer after current line (a new blank line is created).
*O*txt<Esc>	*O*pen the buffer before current line (a new blank line is created).

Special Insert Command Characters You must be in insert mode for these keys to function. These are special characters that may be used when in insert mode only.

Ctrl-@	Repeat last insert if entered as first character of insert.
Ctrl-D	Backtab over autoindent (one shift width); autoindent must be set.
^Ctrl-D	Remove autoindent for current line; autoindent must be set.
0Ctrl-D	Remove autoindent until the next autoindent character (Ctrl-T) is typed.
Ctrl-T	Tab over the amount specified by the shiftwidth.
Ctrl-V	Escape the special meaning of insert characters and allow insertion of nonprinting characters. The Ctrl-J (new-line) character cannot be inserted as a visual character, nor can the Ctrl-I (tab).
Ctrl-W	Erase last word of input.
\	Escape the special meaning of insert characters. The Ctrl-V is a better, more reliable method.
Backspace	Erase last character of input by pressing your backspace character. The default is # but you may have set it to Ctrl-H using **stty**.
Kill	Erase current line of input by pressing your line kill key; the default is @ but you may have set it to Ctrl-U using **stty**.
Delete	Exit insert mode abnormally by pressing Delete.
Esc	Escape insert mode; return to command mode.

Operators Follow an *operator* with any motion command to affect units of text. Enter an operator twice to affect lines. Operators cannot be mixed. The *txt* esc should be used after the *c* OPR MOT command. It is not used on the commands listed (to save space). A number *x* may be placed before or after an operator. If a number, *x*, precedes the operator and another number, *y*, follows the operator (xOPRyMOT) then the numbers are multiplied, $x * y = z$, and the operator function is performed on *z* repetitions of motion (e.g. 5d4d = 20dd = d20d; any of these commands will delete 20 lines of text).

Operator	*Description*
c	Change text.
d	Delete text.
>	Indent (Shift right) text.
<	De-indent (Shift left) text.
y	Yank (Cut/Store) to register.
!	Filter text through UNIX command and replace.
=	Pretty print LISP s-expressions.

x*OPR*n*MOT*[*txt*]	Perform function specified by operator OPR, *x* multiplied by *n* motions MOT. If the operator is an insert-oriented command then insert the txt text and press Escape when finished.

Changing Text Combine *c* with any motion command to *change* units of text. The following list contains aliases or shorthand commands for performing changes and various other text changing commands.

*n*s	*S*ubstitute *n* characters.
*n*cc	*C*hange *n* entire lines (see *S* below).
*n*C	*C*hange the rest of the current line and the next *n* lines.
*n*S	*S*ubstitute *n* entire lines (see *cc* above).
~	Invert current character case and move right one space.
x*cn*MOTtxt	*C*hange text defined by *x* multiplied by *n* motions with new txt text.

Deleting Text Combine *d* with any motion command to *delete* units of text. The following list contains aliases for the d operator and various other deletion-oriented commands. The deleted text is stored in the undo register and if the text includes new-lines, then it is placed in register 1. As more line-oriented deletes are made, the text is shifted to the next higher register. Deleted text may be retrieved from registers using the p command. The last delete can be retrieved using the u (undo) command.

*n*x	Delete the current character and shift text left.
*n*X	Delete the character to the left of the cursor.
*n*xp	Transpose the current character and the next character.
*n*dd	*D*elete the entire current line.
D	*D*elete the rest of the current line (same as d$).
x*dn*MOT	*D*elete text defined by *x* multiplied by *n* motions.

A few examples are:

dL	*D*elete to end of the screen.
dG	*D*elete to end of the buffer.
d1G	*D*elete to the beginning of the buffer.
5dd	Delete 5 lines including the current line.

Replacing Text The following commands are used to replace text. They are not combined with motion commands.

*n*r*c*	*R*eplace current character with character *c*.
*n*R	*R*eplace characters forward on current line until Esc pressed; new lines of text may be inserted.

Indenting Text The following two commands are used to indent and deindent text defined by a certain motion.

x>*n*MOT	Indent text *x* multiplied by *n* shiftwidths from current line to position defined by motion MOT.
x<*n*MOT	De-indent text *x* multiplied by *n* shiftwidths from current line to position defined by motion MOT.

Filtering Text Filter commands take the text specified by the motion and send it to a UNIX filter. The text is deleted from the buffer and replaced by the output of the UNIX filter.

!!*cmd*	Send current line to UNIX *cmd* and replace current line with output of *cmd*.
n!!*cmd*	Send *n* lines to UNIX *cmd* and replace specified *n* lines with output of *cmd*.
!*n*MOT*cmd*	Send *n* motions of text to standard input of any UNIX filter command to be manipulated and then returned; the original text is replaced. A filter command must be standard input and output capable (e.g., grep, sed, sort, etc.) You DO NOT use a colon for this command.

An example is well worth explaining for this command. Assume you have a screen full of text you need to convert into two columns instead of the one you currently have in the buffer. To do this you just type the following command

```
!Lpr -t2
```

and press Return. The !L command takes all of the text from your cursor to the bottom of the screen and sends it to the **pr -t2** command. The **pr -t2** command reformats the input text into two-column output. Your screen is redrawn with the same data in two-column format. If you don't like the way it looks, you can always press u for undo.

Moving Text (cut & paste) By using the y operator you can yank any amount of text based on a motion command. The yank command is referred to as cutting the text in some editors. The p command is used to put the text back in the buffer.

Yanking (cutting) The text may be yanked to the undo register or to a named register.

*n*y*MOT*	*Y*ank *n* motions of text to *undo* register. Text is copied to register.
*n*d*MOT*	*D*elete *n* motions of text to *undo* register. Text is copied to register and deleted from buffer.
"*reg*n*OPRMOT*	Perform *operator* function *n* motions and *store* in an alphabetic register ([a-z]). Where *\s- 2REG\s+2* is a lowercase letter.
"*REG*n*OPRMOT*	Perform *operator* function *n* motions and *append* to an alphabetic register ([A-Z]). Where *REG* is an uppercase letter.

Learning to use the yank command is beneficial if you do much editing. The following examples illustrate how to use the yank command.

5yy	Yank five lines to the undo register.
"ayG	Yank all lines from the current line to the end of the buffer into register a.

Putting (pasting)

p	*P*ut data from *undo* register *after* current line.
P	*P*ut data from *undo* register *before* current line.
"*REG*p	*P*ut lines from register REG *after* current line.
"*REG*P	*P*ut lines from register REG *before* current line.
"[1-9]p	*P*ut 1-9th *last delete* from corresponding number register.

Moving text between files You can move text from one file to another without leaving **vi**. You must first save the text to a named register then switch to the other file, move to the location you wish to place the saved text, and then put it in the buffer.

*REG*y*MOT*	Save the text to a named register (a-z).
:e#	Switch to *alternate* buffer/file in the *vi* session buffer.

- or -

:e *file*	Switch to a new *file* in the *vi* session buffer.
MOT	Move to the desired location in the *alternate* or new file.
*REG*p	*Put* lines from register *after* current line.

MACROS Macros are a special feature of the **vi** editor that allows you to store multiple keystrokes and reference them with a single or shorter set of keystrokes. They are designed to save keystrokes. You can have multiple **vi** or **ex** commands defined in one macro. There are two types of macros, the string macro and the register macro.

NOTE

Macros can be used in **ex** or **vi**. The string macros are set up in **ex** mode. The register macros may be set up in either **ex** or **vi** mode. The descriptions listed below for register macros are using the **vi** mode method because it is easier and more common.

String Macros (Mapping) String macros are stored in the *map* area of **ex**. There are two types of string macros; one is used in command mode, the other is used in insert mode. You create string macros using the **ex** *map* command. You can remap an existing command key to have a new meaning or you can use one of the unused characters.

Command Mode String Macros The following command formats are used to create command mode string macros.

:map *c* act	Creates a command mode character command named *c* that performs the *act* action when pressed. *Act* may be any series of actions to create your own single-character command.
:map #*n* act	Map function key *n* to a specific action.
:map	List all map commands.
:unmap c	Remove *c* from map table.

For example, you may want to edit multiple files from the same command line. Normally, you have to type :n and press Return to edit the next file. You can map a character to perform this function for you. The following key sequence sets the character \ (backslash) to perform the :n<cr> for you.

```
:map \ :n<Ctrl-V><Ctrl-M>
```

The Ctrl-V is used to escape the meaning of the Ctrl-M (Return).

Insert Mode String Macros The following formats are used to create insert mode string macros.

:map! short long	Creates an insert mode macro named *short* that inserts the characters in *long* when it is typed. You can place an Escape character in the *long* string to enter command mode, then return to insert mode.
:map!	List all insert map commands.
:unmap! short	Remove insert *c* from map table.

The string defined by short is interpreted as soon as the last character of the string is entered. For example, if you defined the following macro

```
:map! usv UNIX System V Release 3.2
```

and you enter insert mode and type usv, **vi** immediately expands the string to UNIX System V Release 3.2.

Input Mode Motion Definitions You can make **vi** perform somewhat like a word processor or true screen editor by defining input macros. For example, if you press the Up Arrow key in command mode, the cursor moves up one line. But in insert mode you insert strange characters or return to command mode. In a word processor you press the Up Arrow key in insert or noninsert mode and the cursor moves up a line.

In **vi** you set the definition of your cursor keys to the command that would leave insert mode, perform the motion command, and then return to insert mode. The following table defines the more popular definitions for a vt100 type terminal.

Input Key Symbol	*Input Key Sequence*	*vi Command*	*Function*
Up Arrow	<Esc>OA	<Esc>ka	Move up one line
Down Arrow	<Esc>OB	<Esc>ja	Move down one line
Left Arrow	<Esc>OC	<Esc>ha	Move left one space
Left Arrow	<Esc>OD	<Esc>la	Move right one space

To map the first one of these in **vi** simply type the following:

```
:map ^V^[OA ^V^[ka
```

You will probably want to place these definitions in your .exrc file. The command is the same except you leave off the : (colon).

Register Macros Register macros are stored in named registers. They may be multiple lines in length and consist of **vi** or **ex** commands. They can be used in **vi** or **ex** mode. To invoke a register macro you press @x, where x is the name of the register where the commands are stored. To store commands for use as a register macro, you enter insert mode and type in the **vi** or **ex** commands you wish to execute when the macro is used. Then exit insert mode and delete the lines displayed to the desired named register.

The following commands are used to invoke a register macro.

@*x*	Invoke commands (macro) stored in register *x* while in **vi**.
:@*x*	Invoke commands in register *x* while in **ex**; acts strange on some REs.
@@	Repeat last macro.

Example Macro:

o:%p	Press *o* to enter insert mode, enter the **ex** command :%p (used to print out the entire buffer contents to your screen). Exit insert mode by pressing Esc.
"pdd	Save the :%p line to register *p* and remove it from the session buffer.
@p	Invoke the macro, the :%p command is executed and the entire file is displayed on your terminal screen.

ABBREVIATIONS Abbreviations are almost exactly like the insert mode string macros. They are designed to save keystrokes. The difference between abbreviations and insert string macros is that abbreviation strings are interpreted only if the string is followed by a space, tab or new-line.

:ab abr string	Abbreviate *abr* for *string*. Each time *abr*<space> is entered in insert mode, **vi** automatically replaces it with *string*.
:ab	List all abbreviations.
:unab abr	Unabbreviate abbreviation *abr*.

The following examples may be useful when using **vi**.

:ab hte the Abbreviate *the* each time *hte* is entered in insert mode. A file of commonly misspelled words can be created to provide an automatic spell correction utility based on this use of *abr*.

:ab swbt Southwestern Bell Telephone
Substitute a short acronym for a long name. Then when you enter swbt<space> in insert mode the long name is substituted in its place.

TRANSPOSING TEXT

~	Invert current character case (upper/lower) and move right one space.
xp	Transpose two characters (delete and put).
dwwP	Transpose two words.
ddp (φ)	Transpose two lines.

UNDO, REDO, and RETRIEVE

u	Undo last change.
U	Undo all changes on current line.
.	Repeat last command.
&	Repeat last **ex** substitute command.
[1-9]p	Retrieve 1-9th last delete from a number register.
*REG*p	Retrieve previously stored contents from a named register.
1pu.u.u	Scan the number registers for last deleted text.

MISCELLANEOUS

=*n*MOT	Pretty print *n* LISP s-expressions.
.	Repeat last **vi** command.
J	Join next line to current line.

LEAVING vi

ZZ	Save buffer to disk and quit **vi**.
:wq [*file*]	Save buffer to disk and quit **vi**.
:x [*file*]	Save buffer to disk and quit **vi**.
:q	Quit **vi** if no updates since last save.
:q!	Quit **vi** without checking for unsaved updates.
:sh	Leave **vi**, go to subshell, press Ctrl-D, or enter **exit** to return to **vi**.
Ctrl-Z	Stop editor, place in background, and return to top level shell. Berkeley UNIX only, **fg** command resumes editing.
:	Enter temporary **ex** mode; perform one **ex** command.
Q	Enter **ex** mode; perform multiple **ex** commands until "open" or "vi" is entered at the colon prompt.

SUBSTITUTIONS Substitutions are handled by the **ex** portion of the **vi** editor. Refer to Module 43 for further discussion about **ex** and substituting.

TAGS Tags are discussed in Module 43, which discusses the **ex** editor.

vi QUICK REFERENCE Appendix G contains a quick reference for **vi**.

CUSTOMIZING OPTIONS The **ex/vi** editor has many options that you can set to customize how the editor functions. Please refer to Module 43 for a description of all supported options.

DIAGNOSTICS AND BUGS The **vi** editor uses the terminal definition entries stored in the TERMINFO database. If these entries are corrupted or deleted, **vi** cannot function correctly for the affected terminal type.

CAUTION

If you have saved text in named registers, **vi** will not warn you of this fact when you decide to exit the file. Thus you may lose the only copy of the text since you stored it in a named register. The solution is to always put any text back from your named register before you write (:w) your buffer to disk.

RELATED COMMANDS Refer to the **ed**, **ex**, **sed**, **grep**, **egrep**, and **fgrep** commands described in modules 39, 43, 117, 60, 40, and 52.

RELATED FILES The **vi** editor uses the TERMINFO database files to know what screen control sequences relate to your terminal type. It also keeps a buffer copy of your original file in the /tmp directory. The files usually start with EX followed by a number.

APPLICATIONS

You use **vi** to edit any ASCII text file. It is the most common editor and is distributed on most, if not all, UNIX systems. It can be used to enter program source code, create documents, edit ASCII databases, or any other text file. It has many powerful features that are well worth spending the time to learn. An expert **vi** user can perform a task up to 5 times faster than a novice **vi** user. The more you understand how **vi** works and the shortcuts it offers, the more efficient you will become.

TYPICAL OPERATION

In this activity you use the **vi** command to edit multiple files. Begin at the shell prompt.

CAUTION

Before starting the following activity make sure your TERM variable is set correctly. If you need help deciding the proper type or have problems when you enter the **vi** editor, contact your system administrator.

1. Type **vi .exrc .tags .profile** and press **Return**.
2. Press **i** and enter the following lines of text; press **Return** at the end of each line.

```
The i places you in insert mode.
To escape from insert mode you press the Esc key, but don't do that yet.
First let's enter a few more lines of text.
This should be enough for now. Press the Esc key after typing this line.
```

3. Press **Esc** to exit insert mode.
4. Type **1G** to move the cursor to the first line of the buffer.
5. Type **yG** to yank the text from the current line to the end of the buffer to the undo (unnamed) register. If your cursor remains at the top of the buffer, type **G** to move it to the bottom line.
6. Now type **p** (lowercase) to put the contents of the undo buffer back in the session buffer.

```
The i places you in insert mode.
To escape from insert mode you press the Esc key, but don't do that yet.
First let's enter a few more lines of text.
This should be enough for now. Press the Esc key after typing this line.
The i places you in insert mode.
To escape from insert mode you press the Esc key, but don't do that yet.
First let's enter a few more lines of text.
This should be enough for now. Press the Esc key after typing this line.
```

7. Type **4k** to move up four lines.
8. Type **2j** to move down two lines.
9. Type **3l** (three lowercase Ls) to move to the right three spaces.
10. Type **cw** to change the word *escape*. Notice the $ sign at the end of the word escape. This notifies you of what text you are changing. You are in insert mode until you press the Esc key.
11. Now type **exit** and press **Esc** to exit insert mode.
12. Type **H** to move to the top left corner of the screen.
13. Now type **dL** to delete all of lines from the current position to the last line on the screen.
14. To retrieve the text you just deleted type **u**. The text returns to your screen. Remember, the *u* command only undoes the last text manipulation commands.
15. Now type **PPP** to put three more copies of the deleted text in the session buffer.
16. Now type **4Ctrl-E** to move the screen down four lines. If your version of **vi** does not support counts then press **Ctrl-E** four times.
17. Type **/First** and press **Return**. Notice when you pressed the / the cursor jumped to the last line on the screen and placed the / there. This is the search facility of vi. The ? searches in the reverse direction. Notice after you pressed Return, the cursor appeared at the first occurrence of the word *First*.
18. Type **wwww** to move the cursor to the next word, four times. Notice the cursor stopped on the ' (single quote) of let's. The small letter word movement commands are punctuation sensitive. The uppercase commands ignore punctuation.
19. Type **BB** and watch how the cursor jumps past the ' without stopping.
20. To jump to the end of the current line type **$**.
21. To jump to the beginning of the current line type **0**.
22. Now sort the entire buffer. Type **1G** to go to line 1 of the buffer.

23. Type **!Gsort** and press **Return** to send the contents of the buffer to the UNIX **sort** command. The ! signals **vi** to filter text based on the following motion command. The motion command used is the G, which goes to the last line of the buffer. Notice all of the lines in the buffer are sorted.
24. Now delete all the lines from the buffer by typing **:%d**. The : enters temporary **ex** mode, the % is the alias for line 1,$ (all lines in the buffer), and the d is the delete command. An alternative command is **1GdG**.
25. Now create a *.exrc* type file by typing **i** to enter insert mode.
26. Type the following lines to create a *.exrc* file you can use when you enter **vi**.

```
set wm=8
set ai
map ^Z :sh^V^M
map v :w^V^M:n^V^M
set tags=/u1/ts/mylogin/.tags<Escape>
```

    The first line sets your wrap margin to 8. This causes **vi** to insert carriage returns in your text automatically when you type past the 72 (80-8=72) column. The second line sets autoindent. The third maps Ctrl-Z to :sh<Return>; the Ctrl-V is needed to escape the meaning of the Return. The third line maps v to be :w<Return>:n<Return> which writes your file to disk and advances you to the next file to edit. The last line sets your *tags* file path to your HOME directory with the filename of .tags.

27. Type **:w** and press **Return** to save the buffer to disk.
28. Now type **:so %** and press **Return** to have **vi** read and interpret the commands in the *.exrc* file.
29. To see the two maps you stored in the *.exrc* file type **:map** and press **Return**.
30. Press **Return** to return to the **vi** screen.
31. Type **v** to write this file again and move to the next file, *.tags*.

In this file we create a tags file for some of the standard files you may need to edit from time to time.

1. Type **:arg** to list the current files being edited.

```
:arg
.exrc [.tags] .profile
```

    The brackets around *.tags* indicate that *.tags* is the current file being edited.

2. Type the following lines of text; press **Return** after each line.

```
exrc    /u1/ts/mylogin/.exrc    1
profile /u1/ts/mylogin/.profile 1
tags    /u1/ts/mylogin/.tags    1
```

    The first line defines a label of *exrc* so when a tag of *exrc* is specified the $HOME/.exrc file is edited starting at line *1*. The next two lines perform the same function for the *tags* file and the *profile* files.

NOTE

The tags file must be sorted in alphabetical order or the tags command will not function properly.

3. Save your *.tags* file to disk by typing **:w** and pressing **Return**.
4. Now to edit your *.exrc* type **:tag exrc** and press **Return**. Notice the editor changes the screen and informs you that you are editing the *.exrc* file.
5. To return to the *.tags* file buffer, type **:e#** and press **Return**. The # sign represents the alternate buffer name.
6. Enough editing, exit the editor by typing **ZZ**. **vi** should complain about having another file to edit. Since we don't wish to edit the file, type **ZZ** again or type **:q!** to override the warning signal.
7. Turn to Module 108 to continue the learning sequence.

Module 152
wait

DESCRIPTION

The internal **wait** command causes the shell to wait for processes to complete before executing the next command. It is usually used to wait for background processes to finish executing. The shell itself waits for foreground processes to complete before executing the next command; therefore, you only need **wait** for background processing.

COMMAND FORMAT Following is the general format of the **wait** command.

```
wait [ PID ]
```

BSD (Berkeley)

```
wait
```

Arguments The following argument may be passed to the **wait** command.

PID A specific process ID of a background job. The shell waits for that process to complete before continuing to process the next command.

If no PID is given then the shell waits for all background processes to finish executing before continuing to run the next command.

FURTHER DISCUSSION If you have some processes that take a long time to execute, you may want them to run in the background. You may find that you need the information produced by one of these commands to perform another task. Thus you may want to wait for the background process or processes to complete before continuing. The **wait** command is used to handle this waiting process.

In shell scripts you may have multiple commands running in background mode. If this is the case you can wait for any one process or for all of them to complete. To wait for all of the background processes to complete you use:

```
wait
```

with no arguments. If you wanted to wait for a specific process to complete you use:

```
wait PID
```

where PID is the process ID of a given process. To know what process ID a process has you can use the shell variable $!. For example, if you have more than one process in the background and you need to wait on the second one to complete, you might use the following code:

```
sort -u data > data.sorted &
PID1=$!
lp report &
PID2=$!
sort -u customer > templ &
PID3=$!
spell memo > memo.mis &
wait $PID3          # wait for customer sort to complete
mv templ customer
wait $PID1          # wait for the sort to complete
mv data.sorted data
wait
echo Misspelled words from your memo file:
cat memo.mis
```

RELATED COMMANDS Refer to the **sleep** command described in Module 121.

RETURN CODES The return code of the specified PID is returned via **wait**. Therefore, you can check the status of wait to see if the process actually terminated correctly.

APPLICATIONS

The **wait** command is normally used in shell scripts to stop command execution until a certain process has completed. Once the command or commands have completed, the return code may be checked for proper process completion. Using the **wait** command is often useful when you can perform parallel processing in your shells but you need to guarantee sequential processing at some point in the script.

TYPICAL OPERATION

In this activity you use the **wait** command in a shell script to wait for a **sort** command to finish executing in the background. Begin at the shell prompt.

1. Type **find /usr/spool -name core > /dev/null &** and press **Return**. The **find** command is placed in the background for processing.
2. Now type **wait $!** to wait for the **find** to complete execution. Some systems may finish the background process before you can type the **wait** command. If this is your case feel lucky. But you should still use the **wait** command at times you must guarantee the proper sequence of command execution.

C Shell

Use **wait** without the $!.

---

3. Turn to Module 142 to continue the learning sequence.

Module 153
wall (BSD)

DESCRIPTION

The external **wall** command performs a **write** to all users currently logged on to the system. **wall** differs from a **write** in two ways. First, it reads all of its input from the standard input. Redirection or a pipe may be used. After receiving the entire input (an EOF is encountered) **wall** sends the message to all users. Second, it is a unidirectional utility. It does not imply a response from the recipients. It functions as a broadcasting tool instead of a bidirectional communication tool.

NOTE

The **wall** command is supported on System V as an administrative command.

COMMAND FORMAT Following is the general format of the **wall** command.

```
wall
```

FURTHER DISCUSSION The **wall** message precedes its message text with the following line:

```
Broadcast Message from username
```

It is usually used by the super-user to warn all users of immediate problems. The most common warning is the system is being shutdown within 60 seconds.

The super-user can write to any user's terminal regardless of the permissions set by the **mesg** command.

Some common formats include:

```
wall < mesgfile
```

- or -

```
cat mesgfile | wall
```

DIAGNOSTICS AND BUGS The following messages may be returned by the **wall** command if it encounters problems.

"Cannot send to /dev/ttyXX"
: Wall could not open the user's tty for writing. Possible problems exist with the tty driver or hardware.

"Permission denied"
: The person you are trying to **wall** has denied other users from writing to his/her terminal. This means the recipient has done a **mesg -n**.

"Can no longer write to user"
: The recipient performed a **mesg -n** after you started your **wall** command. This denies you permission to write to the recipient's tty.

RELATED COMMANDS Refer to the **mesg** command described in Module 88 and the **write** command described in Module 161.

RELATED FILES The following files are used by **wall**:

/etc/utmp	Used to locate the destination users. The ttys are also retrieved from this file.
/etc/ttyXX	A tty device file is used to write to the other users.
standard input	The input is read from your terminal unless redirected from a file or piped from a previous command.

APPLICATIONS

The **wall** command is commonly used by the system administrator (super-user) to broadcast warnings to all users currently logged on to the system. It can be used by a user like yourself. This is not done often since not very many users have a need to send message to everyone on the system.

TYPICAL OPERATION

In this activity you use the **wall** command to broadcast a message to everyone logged onto the system. Note you're performing this at your own risk. If you have a group of hard-core, leave-me-alone type users, you may want to skip this exercise. But if you want to display a message on every user's screen, proceed. Begin at the shell prompt.

1. Type **wall** and press **Return**.
2. Type the following text and press **Ctrl-D**. When you press Ctrl-D, **wall** begins to send the message to each user currently logged on to the system.

```
cj> wall
This is a broadcast message of the UNIX Beginners Broadcasting
Foundation. Please disregard this message and get back to your
work immediately.
Ctrl-D
```

   Since you are logged on to the system, the message will be sent to your tty also. This provides for simple verification that the message was sent as expected.

3. Turn to Module 88 to continue the learning sequence.

Module 154
wc

DESCRIPTION

The external **wc** command counts the characters, words, and lines in a file and displays the output on your screen. The word delimeters for **wc** are space, tab, and new-line. The delimeter for a line is new-line. The input may be the standard input, therefore **wc** is a filter.

The features of **wc** are:

- Counts the number of characters in a file.
- Counts the number of words in a file.
- Counts the number of lines in a file.
- Reads from a file.
- Reads from the standard input.
- Writes to standard output.
- Handles multiple files, provides counts for each file read.

COMMAND FORMAT Following is the general format of the **wc** command.

```
wc [ -clw ] [ file_list ]
```

Options The following options may be used to control how **wc** functions.

-c — Count and display only the number of characters contained in the given files.

-l — Count and display only the number of lines contained in the given files. Lines are defined as ending with new-lines.

-w — Count and display only the number of words contained in the given files. Words are defined as ending with spaces, tabs, or new-lines.

If no options are specified, **wc** returns the count for lines, words, and characters. Thus **wc** assumes the -lwc options.

The counts are displayed in the order the options are given. For instance, if you specify -cwl, **wc** displays characters, words, and lines followed by the filename.

Arguments The following argument may be passed to the **wc** command.

file_list — One or more files to be counted. The requested counts will be displayed. Each filename will be displayed on a line by itself, preceded by the appropriate counts.

If no files are given, then **wc** reads from the standard input, which may be a pipe, indirection, or your keyboard.

APPLICATIONS

You use **wc** to show the size of a file. It is commonly used to count how many lines are in a text document. It may be useful when handling flat ASCII files that are databases. Using **wc**, you can list how many entries exist in your database. It is also used to count the entries in a directory. The basic rule is, if it's ASCII text and you want to know how many lines, words, or characters are in a file, you use the **wc** command.

TYPICAL OPERATION

In this activity you use the **wc** command to count the lines in a file and the number of entries in a directory. Begin at the shell prompt.

1. Type **wc /etc/group** and press **Return**. Your screen resembles the following display:

```
cj> wc /etc/group
   12   12   1172    /etc/group
```

   Notice 12 lines and 12 words. The delimeters for words in **wc** are space, tab, and new-line.

2. Type **wc -w < /etc/passwd** and press **Return**. In this situation **wc** is reading from the standard input. The shell has redirected the standard input to be the /etc/passwd file. Since no filename is known to **wc**, no filename is displayed on your screen. See the following display:

```
cj> wc -w < /etc/passwd
   363
```

3. To display the number of entries in your HOME directory, type **ls | wc -l** and press **Return**. Notice the display is different, no filename is displayed. This is because **wc** read from the standard input, thereby not having a filename to display.

4. Turn to Module 29 (SV), Module 20 (BSD) to continue the learning sequence.

Module 155
whatis (BSD)

DESCRIPTION

The external **whatis** command displays the header line of a given command's man page. The header line contains the name of the command, the section of the reference manual it resides in, and a brief description of the command.

The format of the output is *command(N) description*, where N is the section number. Once the name and section have been returned you can run the **man** command to display the entire man page for a specific command. The format of the **man** command is **man** *section command*.

Actually **whatis** is the same as the **man -f** command.

You must know the name of a *command* to utilize **whatis**. If you do not know the name of the command, try the **apropos** command.

COMMAND FORMAT Following is the general format of the **whatis** command.

```
whatis command ...
```

Arguments The following argument may be passed to the **whatis** command.

command The name of the command you want described. You must know the name of a *command* to use **whatis**.

RELATED COMMANDS Refer to the **apropos** and **man** commands described in modules 5 and 87.

RELATED FILES The **whatis** command looks for header lines in the /usr/man/whatis database file. Your system administrator can create this file by executing the **/usr/lib/mkwhatis** or the **/usr/lib/makewhatis** command while logged in as super-user.

APPLICATIONS

The **whatis** command is helpful in deciding whether a command performs the function you think it may perform. One of the problems with **whatis** is the inconsistent format and wording of UNIX man pages and their headers. The **whatis** command displays the header line, but the header line may be misleading to you. Thus it may be difficult to trace the exact command you want to use.

TIP: The word "print" is often used in header lines of man pages to say that the command displays text to the standard output. This is often misleading, since you may think of "print" as sending output to a printer.

TYPICAL OPERATION

In this activity you use the **whatis** command to display the header line of the **vi** and **cat**. Begin at the shell prompt.

1. Type **whatis vi** and press **Return**. The output resembles this:

```
cj> whatis vi
vi(1) - screen oriented (visual) display editor based on ex
```

2. Type **whatis cat** and press **Return**. The output looks like this:

```
cj> whatis cat
cat(1) - catenate and print
```

   Notice the difference between the two descriptions. The vi one is fairly concise and explanatory, but the cat one is vague, if not useless.

3. Turn to Module 87 to continue the learning sequence.

Module 156
whence (ksh)/type (sh)

DESCRIPTION

The internal **whence** command displays how the shell would interpret the given arguments as commands. For example, if you want to know where the **cat** command is being executed from, you type **whence cat** and press Return. The **whence** command is similar to the Bourne shells **type** command. There are three basic types of commands described by **whence**; they are:

- aliases - the value of an alias is displayed
- internals - the command name is displayed
- externals - the PATH is searched and the full path is displayed

COMMAND FORMAT Following is the general format of the **whence** command.

```
whence [ -v ] [ command ... ]
```

Options The only option **whence** recognizes is:

-v Verbose information. The -v causes **whence** to display a more verbose description of each command listed as an argument.

Arguments The only arguments **whence** can use are commands that exist on the system or in the shell.

command Any command that is an alias, internal, or external command.

FURTHER DISCUSSION For more detailed information about how the command is interpreted you can use the -v option. An example of the three types of commands and the -v option follow to clarify the use of whence:

```
cj> whence -v ls
ls is /bin/ls

cj> whence pwd
print - $PWD

cj> whence -v pwd
pwd is an exported alias for print - $PWD

cj> whence exit
exit is a built-in command
```

The Bourne shell's **type** command is not as smart or complex as the **whence** command. It does not have the -v option nor does it know about aliases since the Bourne shell does not support them. The **type** command is implemented in the Korn shell as an alias:

```
alias type="whence -v"
```

that uses the Korn shell's **whence** command.

RELATED COMMANDS Refer to the **which** command described in Module 158 for an equivalent command used on BSD Systems.

RELATED FILES The **whence** and **type** commands write to the standard output.

APPLICATIONS

The **whence** command is used to find the location or source of a command. **whence** is helpful in finding were certain commands are located. By knowing command locations you can rearrange your PATH directories for more efficient command searches. You can assign new aliases to override the default location of the command, thereby creating your own version of an existing command. Many people just want to know were a command resides.

On some of the multiversion UNIX systems running System V and Berkeley 4.3 **whence** can be used to locate where a command is being found by the shell. For example, on a Sun workstation the Berkeley **echo** command is in /bin/echo, while the System V **echo** is stored in /usr/5bin/echo. The **whence** command would allow you to know which **echo** command your PATH points to first (which echo command is executed).

TYPICAL OPERATION

In this activity you use the **whence** command to locate diffent types of commands. If you are not using the ksh skip to the next module. Although some versions of the csh have incorporated this command, so you might try it if you're using the csh. Begin at the shell prompt.

1. Type **whence cat** and press **Return**. Notice the returned message:

   ```
   cj> whence cat
   /bin/cat
   ```

   This shows the **cat** command resides in the /bin directory.

2. Type **whence -v cat** and press **Return** to display more information about the **cat** command. Notice the more verbose output:

   ```
   cj> whence -v cat
   cat is /bin/cat
   ```

   or

   ```
   cat is a tracked alias for /bin/cat
   ```

3. Turn to Module 148 (SV), Module 158 (BSD) to continue the learning sequence.

Module 157
whereis (BSD)

DESCRIPTION

The external **whereis** command locates the binary, source, and manual page files for a given command. The *command* names you provide are stripped of the following:

- any single-character extensions, such as .c, .f, etc.
- the source code control prefix, s.
- any pathname preceding the command.

Once the *command* is preprocessed, **whereis** searches a set of standard directories for the given *command*. The directories are:

/bin	/usr/hosts
/lbin	/usr/local
/lib	/usr/include
/llib	/usr/lib
/etc	/usr/llib
/usr/bin	/usr/man/*
/usr/lbin	/usr/src
/usr/games	/usr/ucb

Some systems search a slightly different set of directories, depending on the directory structure implemented by the vendor.

COMMAND FORMAT Following is the general format of the **whereis** command.

```
whereis [ -bmsu ] [ -SBM directory -f ] command_list
```

Options The following list describes the options and their arguments that may be used to control how **whereis** functions.

-b	Search for only binary files.
-m	Search for only manual page files.
-s	Search for only source code files.
-u	Search for unusual *commands*. A *command* is considered unusual if it does not have a file in each requested type (bms). For example, **whereis -s -u** * displays files located in the current directory that do not have corresponding source code files.
-f	Terminates the directory_list. Thus it signals the start of the *command_list*.
-B *directory_list*	Specifies the directories where **whereis** searches for binary files for the given *commands*. This option can be used to change or limit the search path used by **whereis**.

-M *directory_list* Specifies the directories where **whereis** searches for manual page files for the given *commands*. This option can be used to change or limit the search path used by **whereis**.

-S *directory_list* Specifies the directories where **whereis** searches for source code files for the given *commands*. This option can be used to change or limit the search path used by **whereis**.

CAUTION

The *directory_lists* must be composed of complete pathnames. Complete or full pathnames always start with a /. For example, /usr/ucb is a full path, but usr/ucb is not.

Arguments The following argument may be passed to the **whereis** command.

command_list The names of commands for which you want entries displayed. Each argument must be a command or file found in one of the specified directories or one of the default directories. The supplied *commands* are stripped of leading pathnames, trailing extensions, and source code control prefixes.

RELATED COMMANDS Refer to the **find**, **whatis**, **whence**, **which**, and **man** commands described in modules 55, 155, 156, 158 and 87.

RELATED FILES See the previous table for the list of directories searched by default.

APPLICATIONS

The **whereis** command is useful for locating where certain files are that have the name of the *command* or file you specified. For example, if you specified **cat** as the command argument to **whereis**, the output would consist of a line for the binary executable, manual page, and, if you have on-line source, the source file.

TYPICAL OPERATION

In this activity you use the **whereis** command to locate files related to the **cat** command. Begin at the shell prompt.

1. Type **whereis cat** and press **Return**. After a brief pause the following information is returned.

```
cj> whereis cat
/bin/cat
/usr/man/man1/cat.1
/usr/src/cmd/cat.c
```

2. Turn to Module 155 to continue the learning sequence.

Module 158
which (BSD)

DESCRIPTION

The external **which** command locates given commands based on your current shell environment PATH and aliases. It displays the path of executable commands that would be executed if given as commands. Each argument to **which** is expanded if it is an alias, then searched for along your PATH. The aliases and path values are retrieved from your .cshrc file.

COMMAND FORMAT Following is the general format of the **which** command.

```
which [ commands ]
```

Arguments The following argument may be passed to the **which** command.

commands The names of commands searched for in your PATH and aliases. Only commands that would normally be executed if given as a command are displayed. Aliases are expanded into the original command and searched for as a normal command.

DIAGNOSTICS AND BUGS If an alias contains more than a single word, a diagnostic message is displayed.

RELATED COMMANDS Refer to the **whence** and **whereis** commands described in modules 156 and 157. The **whence** command is a System V equivalent.

RELATED FILES The **which** command reads the ~/.cshrc file to know about aliases.

APPLICATIONS

You will find **which** extremely useful if you write shell scripts. One of the problems with writing a shell script is recreating an existing command name. You should always use **which** before you create a new command. For example, if you decide you want to create a new command to list only .c files, you might name it ls. But before you do, you need to verify that no other command exists with that name. Therefore, you type which ls and press Return. You will find that ls already exists. Thus you should select another name for your command.

Another good use is to locate which copy of a command you are executing. It is not uncommon for someone, even the system administrator, to create a new command without checking first for an existing command with the same name. Thus if you are suspicious of a command you are running because it does not perform like you expect, you can use **which** to find which copy you are executing.

TYPICAL OPERATION

In this activity you use the **which** command to find the location of the **vi** command. Begin at the shell prompt.

1. Type **which vi** and press **Return**. The output resembles,

   ```
   cj> which vi
   /usr/ucb/vi
   ```

   Depending on which system you are on, vi may be found in one of several possible directories.

2. Turn to Module 157 to continue the learning sequence.

Module 159
who

DESCRIPTION

The external **who** command displays who is logged on the system. With no options, **who** lists information about each user that is currently logged on to the system. The **who** command reads the accounting file (/etc/utmp) and returns requested information. The default is the user name, terminal line, and login time. The following table defines all information displayed based on the header that **who** uses to describe each column of its output.

NAME	The user name (LOGNAME).	
STATE	The write permission of the line (or hardware problem).	
	+	The line may be written to by other users.
	-	The line may only be written to by root.
	?	A problem has been detected with the line.
LINE	The terminal tty name (ttyXX).	
TIME	The time you logged in to the system.	
IDLE	How long since you've typed something on your terminal.	
		It's been less than a minute.
	HH:MM	It's been HH hours and MM minutes.
	old	It's been over 24 hours or a boot occurred and no one has logged in on the line.
PID	The process ID of your login process (login shell).	
COMMENT	Any comments related to your line's /etc/inittab entry. This information is placed in the inittab by the system administrator. Common information is the location of the terminal, the associated telephone number, type of terminal, or any other pertinent information.	
EXIT	The system termination and the program exit values.	

BSD (Berkeley)

Some vendors include a command named **w** which is an enhanced **who** command. Its output is a combination of the finger and who commands output.

COMMAND FORMAT Following is the general format of the **who** command.

```
who [ -HTabdlpqrstu ] [ who_file ]
who [ -qn x ] [ who_file ]
who am i
who am I
```

BSD (Berkeley)

```
who [ am i ] [ who-file ]
```

Options The following list describes the options and their arguments that may be used to control how **who** functions.

-H Causes columns headers to be printed. The possible headers are:

NAME STATE LINE TIME IDLE PID COMMENT EXIT

-T Same as the default option -s, except the STATE field is displayed. The STATE field informs you of a terminal's tty write permissions. If a + is displayed in the field, anyone can write to the terminal. If a - is displayed, only root can write to the terminal. Root can write to any terminal regardless of permissions. If a ? is displayed, a problem exists with the tty line. Usually caused by a terminal or line sending noise to the computer or a bad asynchornous I/O board.

-a Causes **who** to function as though all other options where specified. It is equivalent to -bdlpqrstuTH.

-b The date and time of the last system boot is displayed.

-d The processes that have died and not been restarted by **init** are displayed. The EXIT field contains the termination and exit values of the dead process. Often used to determine why a process terminated.

-l The tty lines that have a **getty** waiting for a login are displayed. The NAME field contains a LOGIN entry. No STATE of the line is known.

-p Lists any processes started by **init** that are still active. The NAME field is the name of the program started by **init**. **init** refers to the /etc/inittab file to know which programs to start. The STATE, LINE, and IDLE fields are not defined for this option. The COMMENT field contains the contents of the ID field of the /etc/inittab entry for the program.

-q A quick **who**. Only the names of users currently logged on are displayed in a single line, followed by a line containing the total number of users. This option overrides and cancels all other options.

-n *x* Specifies the number of users to display per line. *x* must be at least 1. This option must be used in conjunction with the -q option.

-r Displays the current run level of the system (i.e. the init process). The termination status, process ID, and the exit status of a process are displayed in the IDLE, PID, and COMMENT fields.

-s Displays the default fields of NAME, LINE, and TIME.

-t Displays the last time the system clock was changed by root using the **date** command.

-u Displays the IDLE field in addition to the default NAME, LINE, and TIME fields. The IDLE field contains the time since any activity has occurred on a tty line.

BSD (Berkeley)

Most **who** commands on BSD systems do not support any options. However, you may find that some BSD systems incorporate some of the System V options. You can usually type **who -z** to get a usage listing displayed to your terminal.

Arguments Only two arguments may be passed to the **who** command.

am i Your user name, terminal tty, and login time are displayed. Some systems display the hostname before your username. Most BSD systems only require that two arguments be present for this function to be invoked. For example, **who are are** is equivalent to **who am i**. Your real user ID and group ID are displayed, even if you have **su**'d to another login account.

file The binary accounting file containing the necessary information that **who** displays. The default file is /etc/utmp. An alternate file is /etc/wtmp. Any file may be used but it must be in the *utmp* format described in section 4 of the UNIX Reference Manual.

BSD (Berkeley)

file The binary accounting file containing the necessary information that **who** displays. The default file is /etc/utmp. An alternate file is /usr/adm/wtmp. This file contains records of all logins since the file was created. If you run **who** with this file as an argument, logins, logouts, and system crashes are listed.

RELATED COMMANDS Refer to the **date**, **login**, **mesg**, and **su** commands described in modules 30, 77, 88, and 127, respectively.

RELATED FILES The following list of files are related to the **who** command.

/var/adm/utmp Contains binary accounting information, for users currrently logged on to the system.

/var/adm/wtmp Contains binary accounting information, for last login for each tty. A history of the last logins.

/etc/inittab The system initialize control table.

/etc/utmp The BSD version of the System V /var/adm/utmp file.

/usr/adm/wtmp The BSD version of the System V /var/adm/wtmp file.

APPLICATIONS

The **who** command has multiple purposes. The most common is to find out who is currently logged in to the system. You may want to know if someone is currently working on the system. The -u option displays how long since they have pressed a key on their keyboard. It can be used to find out when the system was booted. How many tty lines are configured for the system can be displayed by the -l option. The number of users currently logged on to the system can be displayed by using the -q option.

who can be used in shell scripts to find information about a user or terminal. Then decisions can be based on this information. It is a handy and versatile tool that you will find necessary as you use the UNIX System.

TYPICAL OPERATION

In this activity you use the **who** command to display the users currently logged on and their IDLE time. Then you run a quick **who** to see how many users are logged on the system. The BSD version does not support day options. Begin at the shell prompt.

1. Type **who -u** and press **Return**. Notice your screen resembles the following display.

```
cj> who -u
NAME        LINE        TIME            IDLE    PID
bill        tty05       Jan 21 10:21    old     1707
nancy       tty07       Jan 24 08:17    .       1132
mylogin     tty11       Jan 25 08:02    .       28138
tlp         tty17       Jan 25 07:03    5       26324
jill        tty21       Jan 25 09:39    16      29831
```

2. Type **who -q** and press **Return** to display how many users are currently logged on to the system.

```
cj> who -q
bill nancy mylogin tlp jill
# 5
```

3. Turn to Module 56 to continue the learning sequence.

Module 160
whoami (BSD)

DESCRIPTION

The external **whoami** command displays the current effective user name of the terminal session on the tty. It displays who you are logged in as or who is logged in on the terminal in front of you.

The difference between **whoami** and **who am i** is **whoami** displays the correct information even if you have **su**'d. While **who am i** does not work because it uses /etc/utmp. Actually, **who am i** displays your real user name and **whoami** displays your effective user name.

COMMAND FORMAT The format of the **whoami** command follows.

```
whoami
```

RELATED COMMANDS Refer to the **who** command described in Module 159.

RELATED FILES The **whoami** command uses the /etc/passwd file for user information.

APPLICATIONS

You can use the **whoami** command to display who you are logged in as even if you have **su**'d to a new user name. You may want to check your user name before creating files; the **whoami** command assures you of who you think you are or are not.

TYPICAL OPERATION

In this activity you use the **whoami** command to display who you are logged in as on the system. Begin at the shell prompt.

1. Type **whoami** and press **Return**. One line of output is displayed on your screen. It tells you who is currently logged in on the system at the terminal in front of you.
2. Turn to Module 150 to continue the learning sequence.

Module 161
write

DESCRIPTION

The external **write** command sends messages you type on your terminal to another user's terminal. **write** reads the standard input and writes to the specified user's terminal tty.

COMMAND FORMAT Following is the general format of the **write** command.

```
write user_name [ tty ]
```

Arguments The following list describes the arguments that may be passed to the *write* command.

user_name The name of a user who is currently logged in to the system. Use the who command to list who is on the system.

tty Specify which terminal tty to connect to if the same user name is logged on more than once.

FURTHER DISCUSSION **write** begins the conversation by sending the destination user two bells (terminal beeps) and a message stating that you are sending a message to his/her terminal. Once **write** makes the connection, your terminal will beep twice to acknowledge the sending of your message.

If the recipient responds by using **write**, you will receive two bells and a message. As the other user enters lines of text they will appear on your terminal screen. At this point you can both send and receive lines of text from each other.

Problems may arise if you both don't adhere to a telephone protocol; one listens while the other one speaks, a.k.a. common phone courtesy. If you both send messages at the same time, your screens will become garbled with your text and the other user's messages that are arriving on your screen.

It is advisable to establish the following protocol if you plan on having a bidirectional conversation.

1. You write to a user but don't write any message lines.
2. Wait for the user to respond with write; the recipient should not send any message lines at this point.
3. Now you begin by transmitting your message. You may have already typed the first line; if so just press Return. You continue sending message lines. After you have finished your message, end it with "(o)" for "over" to signal the other person may begin a reply.
4. When you are ready to terminate the session, enter "(oo)" for over and out.
5. To terminate the session enter Ctrl-D (EOF) or Delete.

The protocol does not have to be followed, although it is suggested for conversations. If you only want to send a brief one- or two-line message, then you can perform the write command and enter the text before the other user has to respond. You can also place the message you want to send in a file and redirect the input from the file to **write**. Here is an example of using redirection:

```
cj> write tlp < tmpmesg
```

This will **write** the entire contents of the tmpmesg file to user tlp.

The following display shows a conversation between two users on the system. Assume your terminal is the left column and the user named bill is viewing the right-hand column from his terminal. The bold print is what you type; normal type is what you receive when Bill types a message. Periods at the far left and right of the display mark filler lines added to the display to make it more understandable.

```
From your terminal tty11          -to- Bill's terminal tty05

cj> write bill                    | bill>
.                                 |
.                                 |   Message from mylogin tty11 ...
.                                 |
                                  | write mylogin
   Message from bill tty05 ...    |
                                  |
Hi Bill,                          | Hi Bill,
Lunch at 11:00 (o)                | Lunch at 11:00 (o)
.                                 |
Sounds good to me!                | Sounds good to me! (o)
.                                 |
Ok, see you then. (oo)            | Ok see you then. (oo)
.                                 |
Ctrl-D                            | EOT
cj>                               | Ctrl-D
                                  | bill>
```

If a user is logged in more than once, you must specify to which tty you wish to send your message. If you do not specify the tty, **write** will assume you want to send to the first user listed in the /etc/utmp file. **write** notifies you of the situation by displaying the following information:

Assume nancy is a port hog and is logged in on tty07 and tty14.

```
cj> write nancy
nancy is logged on more than one place.
You are connected to /dev/tty07
Other locations are:
/dev/tty014
```

You may grant or deny other users write permission to your terminal by using the **mesg** command discussed in Module 88. You can toggle between the two permissions. **Mesg -n** denies access and **mesg -y** grants access. The super-user can force messages to a terminal that has been set to deny messages.

Some commands perform a **mesg -n** when they start, to prevent other users from writing to your terminal while they execute. The **pr** and **vi** commands are the most common of these commands.

You can escape from **write** to perform other commands. If you type a ! (exclamation mark) at the beginning of the line, **write** will interpret the rest of the line as a command. The shell is executed first and the command is passed to the new shell for interpretation and execution.

DIAGNOSTICS AND BUGS The following list of messages are produced by **write** depending on the given situation.

"user is not logged in"
: The user you want to write to is not logged on to the system. Execute the **who** command to find out who is logged on.

"Permission denied"
: The person you are trying to **write** has denied other users from writing to his/her terminal. This means the recipient has done a **mesg -n**.

"Warning: cannot respond, set mesg -y"
: You have your message permission set to no. The recipient cannot respond (write) to your terminal. Change your permission by typing **!mesg -y**.

"Can no longer write to user"
: The recipient performed a **mesg -n** after you started your **write** command. This denies you permission to write to the recipient's terminal.

RELATED COMMANDS Refer to the **mail**, **mesg**, **ksh**, **talk** and **who** commands described in modules 86, 88, 82, 132, and 159 respectively.

RELATED FILES The following files are used by **write**:

/etc/utmp	Used to locate the destination user. The tty is also retrieved from this file.
/etc/ttyXX	A tty device file is used to write to the other user.
standard input	The input is read from your terminal unless redirected from a file or piped from a previous command.

APPLICATIONS

The write command is used to communicate with fellow users on an interactive basis. It is useful to send a brief message to someone immediately or to carry on conversations using terminals. The **talk** command is a much more sophisticated utility for having interactive conversations, while **write** serves you better when only needing to send a quick message to a user's terminal. If you want to send someone a note, letter, or a file and they are not logged on, you should use the **mail** command.

TYPICAL OPERATION

In this activity you use the **write** command to send a string of text to the user name *tlp*. Begin at the shell prompt.

1. Type **echo "How about meeting in the small conference room @ 9:00" | write tlp** and press **Return**. **echo** will send the message to **write** via a pipe. The message will appear on the user screen all at one time as shown in the display:

(On tlp's terminal screen)

```
tlp>
   Message from mylogin tty11 ...

How about meeting in the small conference room @ 9:00
```

2. Turn to Module 88 (SV), Module 153 (BSD) to continue the learning sequence.

Appendix A
TERMS AND DEFINITIONS

Term	*Definition*
address	A number specifying a location in the computer's memory. May also specify a line number or range of line numbers when used inside a line editor.
alias	A single name used to reference one or more commands.
ANSI	American National Standards Institute. An institution formed to set standards. One example is the ANSI standard definition for a terminal.
append mode	A mode in a text editor which allows you to append text after the current buffer position.
ASCII	American Standard Code for Information Interchange. Defines the bit sequence codes used to represent control characters, letters, numbers, and symbols on a computer system.
argument	The part of a command line specifying the data to be used by the command. A command may or may not use arguments. An argument may consist of letters, numbers, or text strings. An example is *cat -v myfile*. The argument is *myfile*, the command is *cat*, and *-v* is an option.
array	A consecutive set of data elements referenced by the same name. Each element is referenced using subscripts to index the location in the array.
background	A state of execution referring to commands that have been started from your terminal but are not interactive. The commands execute and are associated with your terminal but do not require input from your terminal. This allows you to continue executing other commands from your terminal.
baud rate	The transfer rate of data from a computer to a peripheral device or from one device to another. Modems and terminals are two common peripheral devices that use BAUD ratings for data transfer.
binary file	A file composed of bits. Usually special application files or compiled machine executable programs are binary. This is more efficient for disk usage and for execution speed.
bit	A binary digit. May be a 0 or 1. Binary means base two numbers.
block special file	A device driver that preforms I/O (reads and writes) data blocks at a time. A disk drive driver is usually a block special file.
boot block	A logical block of information stored on magnetic media (usually disk) containing machine instructions used to boot the system.

Term	*Definition*
branching	A decision point in a computer program that changes the logical flow of a process, program, or structure.
buffer	A temporary storage area, usually in the computer's memory, used by programs to perform data manipulations. For example, the vi test editor copies the contents of your file to a buffer in the computer's memory. Any changes you make to the file affect the buffer image of the file, not the disk copy. When you save your changes the buffer is copied back to disk.
built-in command	A command processed within another program. The shell has many built-in commands.
built-in editors	Editors within the **ksh** and **csh** used to edit the command line. The **ksh** has two, **vi** and **emacs**. The **csh** has a psuedo version of **ed**.
byte	A computer character composed of either seven or eight bits. The number of bits depends on the hardware architecture.
character special file	A device driver that performs I/O (reads and writes) on a character-by-character basis. Terminals are usually a character special device.
child directory	See subdirectory.
child process	A process spawned (started) by another (parent) process.
command	A request for the shell to perform. A command may be a built-in command, an alias, function, program, variable assignment, or programming construct.
command interpretor	A program that reads commands, interprets the input, translates the input commands to instructions understood by the system. Thus the shell is a command interpretor that acts as an interface between you and the computer.
command mode	A mode in a text editor where the characters you type are interpreted as editing commands. This mode allows you to perform certain actions within the editor.
command substitution	The replacement of part of a command with the output of another command.
compiler	A computer program that converts source code written in a high-level computer language to a machine-executable binary file. Once the source code is compiled you execute the new program by entering the filename as a command.
concatenate	To combine separate elements into one. You concatenate files when you combine multiple files into one.
constant	A value used in computer programming languages that remains constant throughout the entire program.

Term	*Definition*
control character	An ASCII character with a value from octal 000 to 037. Control characters usually have special meanings. For example, Ctrl-S is often used to stop text from scrolling off your screen and Ctrl-Q is used to restart the text scrolling.
CPU	Central Processing Unit. The part of the computer that reads machine code instructions and performs the associated task. The Intel 80286 and 80386 are microchip CPUs.
current directory	See present working directory.
cursor	A rectangle or underscore displayed on your terminal indicating where you may enter or delete characters on your display.
daemon	A program usually started when the system enters multiuser mode and remains running to service related user programs' requests. For example, the print spooler daemon processes all print requests queued by the **lp** (**lpr** on BSD) command.
default	The value or condition automatically assigned (assumed). You must explicitly change the value or condition to keep the default from being used in most situations.
delimiter	A character or characters that logically separate words or fields of data. For example, a space and tab are used to delimit options and arguments on a command line.
device	A physical unit used by the computer for a specific task. Examples are printers, hard disks, floppy disks, modems, and tape drives.
device driver	A special program written to control a specific device. Device drivers on UNIX are accessed as files. When the file is accessed the major and minor numbers of the device actually map to specific system calls used to execute the appropriate code for the device.
diagnostic program	A program used to isolate problems on a computer.
diagnostic message	A message displayed by a program when an error has occurred.
directory	A file containing the names and inode numbers of other files. The other files may be any type of file (special, directory, or regular files).
disk	A type of magnetic storage device, usually containing several hard disks resembling phonograph albums. This media provides large amounts of data storage with fast access capabilities.
EBCDIC	The standard used by IBM to define what set of binary digits (bits) represent a single character. The code defines alphabetic characters, numerals, control characters, and special characters.
EOF	End-of-file. The Ctrl-D character is used to indicate the EOF mark from your keyboard. But UNIX does not actually use any character to mark the end of a file on disk. Instead the byte count of the file is used.
EOL	End-of-line. The end of a line marker. UNIX usually uses a new-line.

Term	*Definition*
electronic mail	The exchange of information via a computer and other electronic devices. UNIX provides **mail/mailx** for reading and writing mail messages.
environment	The conditions which control your computer processes while you are logged in to the system. Your environment on UNIX allows you to customize your login shell.
environment file	The file executed each time the shell begins execution. Only used by **ksh** and **csh**.
erase character	The character used to delete the previous character you just typed on your terminal. UNIX allows you to reset this to any character using the stty command.
escape	The ability to leave a program and then return at a later time at the same point from which you exited, such as in the vi editor when you type :sh<CR>.
execute	The act of processing machine instructions by the computer. A program is loaded into memory, then the computer's CPU begins operations based on instructions within the program.
executable file	A file containing instructions that are processed by the system. The program is stored in a file. UNIX allows shell scripts to be executable files. When you enter the name of an executable file UNIX checks the contents then executes the file accordingly.
export	To pass variables to a child process.
external command	A standard UNIX command requiring a process ID for execution, as opposed to an internal command which executes within the current process.
FIFO	See queue.
file	A collection of data. A sequence of bytes. Files may contain data, programs, or various other text. Data is read from and/or written to the file. The file may be an executable script or program.
file descriptor	A number associated with an open file.
file system	A large logical structure used to store boot blocks, inode blocks, and data blocks. All files are stored in data blocks within the file system.
filename	A sequence of characters used to identify a file. The total number of characters must not excede 14 characters in System V. A slash (/) cannot be used in UNIX filenames.
filter	A program that reads from the standard input and writes to the standard output. The standard input of the filter may be your terminal keyboard or the standard output of another program. The standard output of the filter may be sent to your terminal screen or the standard input of another program.
firmware	Executable machine code stored in EPROMS, PROMS, or ROMS. Usually used to perform initial boot procedures.

Term	*Definition*
flag	A command option. See option.
foreground	The execution of one command from your terminal. UNIX executes the first command. After the first command completes, the computer prompts you for another command.
fork	The act of creating a new process. The new process is a child process of the parent which forked it. There is a system call used to spawn (start) a child process.
full-duplex	A type of data communications where the computer can transmit and receive data simultaneously. Your terminal usually communicates with the UNIX System in full-duplex mode.
full pathname	A pathname that begins with the root directory (/) and continues to the specific directory or file.
function	A list of commands referenced by a command name. A function is user definable. It is stored in a file and loaded into the shell's internal memory. When you reference the function's name the function is executed.
global	Applies to the entire file or environment. For a file, global is used in text editor terminology, referring to lines 1 to EOF. In the environment, global refers to variables that are known throughout the environment.
group ID	A number relating to a specific group of users. Used by the system and utilities to control who has access at the group level.
hardware	The physical components combined to create a part of a computer or the complete system.
here-document	Lines of input redirected by the shell to a program. The input immediately follows the program and is read by the shell and written to the standard input of the program.
hidden character	A character that is not displayed on your terminal. This is usually a control or special character.
history file	The file used by the shell to save each command you entered at the prompt.
input/output	The input and output of data by the computer to and from its associated devices.
input/output device	A physical electronic device that can receive input from another device and/or send output to another device. Your keyboard is an input device and your terminal is an output device.
inode	Information node for a file. Contains all pertinent information concerning a file.
inode number	The number of a file's inode contained in the second field of a directory entry.
insert mode	A mode inside a text editor. The characters you type in this mode are entered into the editor's buffer as text.

Term	Definition
interactive	An operating system that you interact with moment by moment. You enter a command and wait for a response. When the program resonds back, you enter further information or another command.
interactive program	A program that communicates with its user. The program requests information and the user responds appropriately. Usually the input device is a keyboard or mouse and the output is to a terminal screen or printer.
internal command	Same as a built-in command.
interpreter	A program that reads commands and executes them.
interprocess communication	The ability for two or more separate programs to communicate with each other.
keyword parameter	The assignment of a value to a variable on a command line. Thus the variable is only known by the command being executed on the command line.
keyword command	A word in the shell that has special meaning. The keywords in the UNIX shell perform certain program control.
line editor	A program that allows you to edit text line-by-line. Still useful in shell scripts.
library	A collection of commonly used utility programs. Most computers have source code, object code, and compiled program libraries. This eliminates the need for programmers to write commonly used programs.
login	The procedure you use to gain access to the computer system.
login name	See user name.
metacharacter	A character that has special meaning to a computer program. The shell uses metacharacters to perform filename matching and expansion.
modem	An electronic communications device used to transmit data accross phone lines.
multitasking	A computer operating system that allows multiple tasks (processes) to execute simultaneously. Actually, the execution of each task is serial. The operating system shares the CPU with multiple tasks by swapping the tasks in and out of the CPU.
multiuser	A computer operating system that allows multiple users to use the system's resources simultaneously. A multiuser system implies that it is multitasking.
operating system	The software on a computer system that is loaded at boot time. All other software is controlled by the operating system. All computer resources (devices, memory, etc.) are also controlled by the operating system.
option	Special arguments on a command line that control how the program will function. On UNIX the convention for specifying an option is to precede it with a dash (-).
ordinary file	A file containing text or data.

Term	*Definition*
output	The writing of information from the computer to an output device such as a printer, terminal, or even a file.
parameter	A variable or constant placed on a command line. Options and arguments are command parameters.
parent directory	The directory directly above a directory. Referred to relatively as "..".
parent process	A process that spawned (started) a new process. The new process is referred to as a child process.
parity	A checksum technique used to varify proper transmission of data. Even parity adds all the 1 bits in a byte and sets the parity bit to 1 or 0 to make the sum of all 1 bits even.
parse	The process of scanning and deciphering data based on certain criteria. A program parses its command line deciding which are arguments and which are options.
path	A list of directories searched for specific types of files.
pathname	Directory names separated by slashes (/) preceding a file or directory name. See full path and relative path.
peripheral device	Devices controlled by the main computer. Used to input, output, and store data. Terminals, printers, disk drives, and modems are all peripheral devices.
permissions	Modes on directories and files that control which users can access certain files for reading, writing, and executing. The **chmod** command is used to change permissions.
pipe	The process of sending the output of one command to the input of another. The shell creates a pipe when a "|" character is encountered.
pipeline	Multiple filters separated by the pipe character "|." The output of the first filter becomes the input of the next.
ports	The physical devices where external peripheral devices are connected to the computer. Terminals and printers are the most common type of peripherals connected via physical ports.
positional parameter	The parameters (arguments) of a shell script command line that have been assigned to variable names beginning with 0 and continuing through the number of arguments on the command line.
present working directory	The directory in which you are currently located. The dot (.) is used to represent the current directory.
primary prompt	See prompt.
printer	A physical device used to print data on paper or some other physical media.
process	A program in execution. Includes the contents of memory, register values, files statuses, current directory, and login related information.

Term	Definition
program	A set of instructions that can be executed by a computer to perform specific tasks.
prompt	A string displayed on your terminal to inform you the shell is ready to accept the next request. The value of the prompt is stored in the variable PS1 (ksh) and prompt (csh).
queue	A structure to control the processing of data or programs. The first data in the queue is processed first. Often refered to as a FIFO, first-in-first-out.
RAM	Random Access Memory. Used to store programs during execution. Allows direct access to any memory location.
read-ahead	The ability to read input while processing output or previous input.
regular expression	Any string of text used to search for matching text in a file. Special regular expression characters allow various matching patterns.
relative pathname	A pathname to a directory or file starting from your present working directory.
remote system	Any system other than the one on which you are currently working.
ROM	Read only memory. Once the memory has been loaded with instructions it can only be read from by other programs.
root	The top level directory of the UNIX file system hierarchy. Designated by a slash. Also the user name of the super-user.
screen editor	An editor which edits text relative to the position of the cursor. The changes are visually updated as you make them.
script	A program written in a shell language.
secondary prompt	The string displayed by the shell if the command you typed at the primary prompt was incomplete.
shell	A program that reads requests, interprets them, and requests tasks from the operating system. It is an interface between you and the computer.
shell script	See script.
signal	A message which is sent from one process to another.
source code	The English version of a program. The source code must be compiled into object code and then linked into machine code.
special character	See metacharacter.
special file	A file used to interface with a device. Referred to as device drivers in the kernel. The /dev/ttyXX files are special files used to communicate with your terminal or printers.
spooler	A subsystem of the computer system which stores print requests on disk until the printer is available for printing.

Term	*Definition*
standard input	The default file assigned to file descriptor 0. Usually this is set to /dev/ttyXX for your shell to read input, where XX is the number of the tty on which you are logged in to the system.
standard output	The default file assigned to file descriptor 1. Usually this is set to /dev/ttyXX for your shell to write output.
standard error	The default file assigned to file descriptor 2. All programs display their messages to this file descriptor.
stream	A sequence of input data to or output data from a program.
string	A set of characters. Any pattern of characters combined together, which may include metacharacters.
subdirectory	A directory contained within another directory. If you list the current directory, the subdirectory is displayed as a file.
subscript	A numerical expression contained within brackets ([ ]) after a variable which evaluates to a specific index within an array.
subshell	A shell started (spawned) by a parent shell.
super block	The second block of a file system which contains critical information about the file system.
super-user	The "root" login. The super-user is a user who has operating system authority to perform any task on the system. Equivalent to other systems' "supervisor" level users.
system	A complete and functional computer system. Usually refers to the hardware and the software as a whole.
system administrator	The person in charge of systems analysis, operations, maintenance, and various other administrative tasks on a UNIX operating system.
task	A unit of work to be performed by the computer.
terminal	A device used to input data to and display output from a computer system. It allows you to interact with the computer.
timesharing	A computer operating system which allows multiple users to share the resources of a computer. It appears to user that the resources of the computer are dedicated to him.
tty	Teletype. A device resembling a typewriter that provided interactive access to the computer. The predecessor of the terminal.
UNIX System	A time sharing interactive computer operating system and the associated application and language software.
user	A person who uses a computer or computer operating system.
user ID	A number assigned to each user for identification by the system and utilities.
user name	A string of characters used to identify a user on the computer. Each user name on a system must be unique.

Term	*Definition*
utility	A software program used to perform routine tasks.
variable	A symbolic name whose value may change throughout a program.
working directory	See present working directory.

Appendix B
UNIX EXERCISES

1. About This Book
   a. What part of UNIX manages the computer system's resources?
   b. What function does the ksh perform?
   c. What 5 requirements do you need to fill before using UNIX?
2. A Sample Session with UNIX User Commands
   a. Which command do you use to reset your "kill character" and "Backspace" keys?
   b. Name the three types of standard I/O.
   c. How do you return to your HOME directory?
   d. What keys abort a program?
   e. List 10 common commands in UNIX.
3. System Overview
   a. What is the basic philosophy of the UNIX Operating System?
   b. What does multitasking mean?
   c. What does multiuser mean?
   d. What are four characteristics of the UNIX file system?
   e. What are the three types of files used in a UNIX file system?
   f. What two functions does the shell perform?
4. alias/unalias
   a. What is the purpose of the **alias** command?
   b. What is the difference between **csh** and **ksh** aliasing command formats?
5. apropos
   a. Why might you use the **apropos** command?
   b. Write the command to find information about line editors on UNIX.
6. awk/nawk
   a. List three uses of the **nawk** program.
   b. List five features of **nawk**.
   c. What does the **nawk '{ print }' file** command do?
   d. Write a program to print a header, the /etc/group file, and a footer.
   e. How would you search for the pattern "tom" in the first field of input?
   f. What is the syntax for accessing the 10th element of an array named NAMES?
   g. Write the **nawk** script to display the first and fifth fields of the /etc/passwd file. Field one should be left justified in a 10-column output field. Field five should be left justified in a 40-column output field.

7. banner
   a. What is the function of the **banner** command?
   b. What options are used with the **banner** command?
   c. What is one common use of the **banner** command?

8. basename
   a. What functions are performed by the **basename** command?
   b. Why might you use the **basename** command in a shell script?

9. bg
   a. What is the purpose of the **bg** command?
   b. Write the command to place job 2 into background processing.

10. cal
   a. What does the **cal** command produce?
   b. What does the command **cal 99** produce?

11. calendar
   a. What service does the **calendar** command provide?
   b. List the date formats recognized by the **calendar** program.
   c. What does the - option cause to happen?

12. cancel
   a. What two functions are performed by the **cancel** command?
   b. Why might you use the **cancel** command?
   c. Write the command line to cancel print job "imagen-915."

13. cat
   a. List six uses of the **cat** command.
   b. How can you stop and start scrolling?
   c. Write the command to combine file1 and file2 into file3.

14. cd
   a. What is the purpose of the **cd** command?
   b. Write the command to change directories to your parent directory.
   c. Write the command to return to the previous directory.
   d. Write the command to change to the user "njf" HOME directory.

15. chfn
   a. Why might you use the **chfn** command?
   b. What information is changed by the command?
   c. Write the alternate format of the **chfn** command?

16. chgrp
   a. What is the function of the **chgrp** command?
   b. Why might you use the **chgrp** command?
   c. Write the command to change all files in your HOME directory to the "mkt" group.

17. chmod
    a. What is the purpose of the **chmod** command?
    b. Write the symbolic mode command to change the permissions on file1 to read and write for all users.
    c. Write the previous command using absolute modes.
18. chown
    a. What is the function of the **chown** command?
    b. List two reasons for using the **chown** command.
    c. Write the command to change the ownership of the present working directory to "tina."
19. clear
    a. What is the function of the **clear** command?
    b. Why might the **clear** command not work properly?
20. cmp
    a. What is the purpose of the **cmp** command?
    b. What is the return code if the files are identical?
    c. Write the command to compare the standard input to file1.
21. col
    a. What is the function of the **col** command?
    b. When might you use the **col** command?
    c. Write the command to use the **col** command with the **cat file1** command.
22. comm
    a. List three functions the **comm** command performs.
    b. Write the command to display the lines only found in the second file.
    c. Explain what text is displayed by the **comm -3 file1 file2** command.
23. Commands
    a. What is the format for a simple UNIX command?
    b. What signifies a parameter to be an option instead of an argument?
    c. What must precede an option or group of options?
    d. What does a dash preceded and followed by blanks signify to a command?
    e. What are the two types of commands in UNIX?
    f. What is the normal return code for a command that completed successfully?
24. cp
    a. List the four uses of the **cp** command.
    b. Define source file and destination file.
    c. Explain the command **cp file1 /etc/tty**.
    d. Explain the command **cp file1 file1**.
25. cpio
    a. What are the functions of the **cpio** command?
    b. What other UNIX command is often used with **cpio** when working with directory structures?
    c. Write the command to copy all of the files in your HOME directory to the /tmp directory.

26. crontab
    a. What is the purpose of the **crontab** command?
    b. Write the cron command line to execute a command on Mondays and Fridays at 10:00 A.M., 2:00 P.M., and 4:00 P.M.
    c. Where is the standard output from your crontab commands sent if you do not redirect it?
    d. Write the commands required to create a crontab file and place it in the cron directory.

27. crypt
    a. What is the function of the **crypt** command?
    b. Can "root" (the super-user) read your encrypted file?
    c. Write the command to encrypt file1. You want to be prompted for the encryption key.

28. csplit
    a. What does the **csplit** command do?
    b. What are the uses of the **csplit** command?
    c. Write the command to split file1 at each occurrence of " (.*)."

29. cut
    a. What two modes of cutting are performed by **cut**?
    b. Write the command to cut fields 1 and 5 from the /etc/passwd file.
    c. What causes the error message "line too long" to be displayed?

30. date
    a. Can you (a normal user) change the system date?
    b. What is the default output format of the **date** command?
    c. How do you display only the hours and minutes? (SV only)

31. dd
    a. What are the two main **functions** of the **dd** command?
    b. List five types of conversions the **dd** command performs.
    c. Write the command to convert ASCII input to EBCDIC output.

32. df
    a. What is the purpose of the **df** command?
    b. Write the command to display only the /usr filesystem information.

33. diff
    a. What does the **diff** command do?
    b. How do you make **diff** ignore blanks in the input?
    c. Explain the command **diff - file2**.

34. dircmp
    a. What is the function of the **dircmp** command?
    b. Explain the output of the first listing generated by **dircmp**.
    c. What command makes the output listings easier to read on your screen?

35. dirname
    a. What is the purpose of the **dirname** command?
    b. When might you use the **dirname** command?

36. dirs
    a. What does the **dirs** command display?
    b. Which shell supports the **dirs** command?
    c. What command must be used before the **dirs** command becomes useful?

37. du
    a. What is the purpose of the **du** command?
    b. How do you display only the summary for all files and directories in your present working directory?

38. echo
    a. What are 5 uses of the **echo** command?
    b. Does the BSD version support escape sequences?
    c. What is equivalent to the -n option on the System V version?

39. ed
    a. How does **ed** differ from a full screen editor like **vi**?
    b. Write the command to edit/create file *myfile*.
    c. What are the two modes of operation within **ed**?
    d. What is a regular expression?
    e. Write the global search command to locate all empty lines and display the line number.
    f. How do you enter text into *myfile*?
    g. How do you return to the **ed** prompt from insert mode?
    h. How do you display all lines within the buffer?

40. egrep
    a. What is the function of the **egrep** command?
    b. In what situation will a given pattern not be located?
    c. How do you display lines that do not match the given string?

41. env
    a. What are the two purposes of the **env** command?
    b. What happens if no arguments are given on the **env** command line?
    c. Write the command to start a new **sh** without passing the current environment.

42. eval
    a. What does the **eval** command force the shell to do?
    b. Explain the command **eval echo '$'$X**, where X=FILE.

43. ex
    a. What is the function of the **ex** command?
    b. How do enter full screen mode?
    c. How do you return to **ex** mode to perform multiple commands?
    d. Write the command to mark a position in the buffer as mark "a."
    e. What address command returns you to mark "a"?
    f. Write the command to change all occurrences of "form" to "from" and have the editor request confirmation before making the change.
    g. Explain the command **!$sort -u**, assuming you are on the first line of the buffer.
    h. How do you write the current line to a file named "hold.one"?

44. exec
    a. What are the purposes of the **exec** command?
    b. Write the command to overlay the current shell with the **csh**.
45. exit/logout
    a. What does the **exit** command do?
    b. What keystroke performs the same function?
    c. What does **logout** do that **exit** does not?
46. expand/unexpand
    a. Explain the **expand** command.
    b. How do you convert tabs to only four spaces?
47. export
    a. What is the purpose of the **export** command?
    b. What does the **export** command do if you run it without arguments?
    c. Why is the **export** command important?
48. expr
    a. What are the four types of operations handled by **expr**?
    b. Write the command to delete the first letter from the value of $X.
    c. Write the command to return the length of a string in $STR.
    d. What commands in the **ksh** have replaced the **expr** command?
49. false
    a. What is the function of the **false** command?
    b. What is the return code of the **false** command?
50. fc/!
    a. Describe three uses of the **fc** command.
    b. What happens when you press **Escape** and type a **v**?
    c. How do you execute the last command that began with "ls"?
    d. What does the **!!** command do?
    e. Write the command to execute the last command that started with "vi."
51. fg
    a. What is the purpose of the **fg** command?
    b. Which shells and operating systems support the **fg** command?
    c. Write the command to bring job 3 to the foreground.
52. fgrep
    a. Why might you use the **fgrep** command instead of **grep**?
    b. How do you search for uppercase and lowercase letters simultaneously?
    c. What option prevents diagnostic messages from being displayed?
53. file
    a. What does the **file** command do?
    b. What type of data within files may confuse the **file** command?
    c. Why might you use the **file** command?

54. Filenames and Filename Generation
    a. List five *de facto* standard filename extensions.
    b. Why would you not want to use the ">" character in a filename?
    c. What are some of the files matched by the filename notation "?ite"?
    d. Why are you unable to use the "/" character within a filename?
    e. How do you perform filename completion using the **ksh**?
    f. How do you perform filename completion using the **csh**?
55. find
    a. What does the **find** command do?
    b. Write the command to display all the files in your HOME directory and all subdirectories.
    c. Write the command to remove all files which have not been modified in 30 days, starting at the current directory.
56. finger
    a. What default information is displayed by the **finger** command?
    b. What information is placed in the .project file?
57. fmt
    a. What function does the **fmt** command perform?
    b. Write the command to format file1.
    c. Write the command to format the next paragraph inside **vi**.
58. fold
    a. What does the **fold** command do?
    b. Describe some uses of the **fold** command.
59. getopts
    a. What is the purpose of the **getopts** command?
    b. What are valid options for **getopts -x:**?
    c. If an argument is required after an option, where is the argument stored?
60. grep
    a. What does the **grep** command do?
    b. What does the name "grep" mean?
    c. Write the command to locate all lines beginning with "r" in the /etc/passwd file.
61. groups
    a. What does the **groups** command do?
    b. Explain the difference between your primary group and secondary groups.
62. hash
    a. What is the purpose of the **hash** command?
    b. Can the hashing function be automatic; if so how?
63. head
    a. What is the function of the **head** command?
    b. Why might you want to use it?
    c. What does the command **head +25 afile** do?

64. history
    a. What specific file does the history command display?
    b. How do you control the number of commands stored in the history file for the **ksh**?
    c. How do your change the history character to a "." from "!" in the **csh**?

65. hostname
    a. What does the **hostname** command do?
    b. What are some uses of the **hostname** command?

66. id
    a. What information is displayed by the **id** command?

67. Input/Output and Redirection
    a. What are the three standard types of I/O?
    b. Write the command to force the standard error onto the standard output, for **ksh** and **csh**.
    c. How do you assure input is read from your keyboard by the **ksh**'s **read** command?
    d. Write the command to redirect the standard output to file $X.

68. jobs
    a. Which shells support the **jobs** command?
    b. What is the purpose of the **jobs** command?

69. join
    a. What is the function of the **join** command?
    b. Write the command to join file1 and file2 on key column 3.

70. kill
    a. What does the **kill** command do?
    b. What command is often used to provide PID information so you can use the **kill** command?
    c. Why might you use the **kill** command?

71. ksh/csh/sh
    a. What two major functions does the Korn shell provide?
    b. List 7 enhancments the **ksh** provides that the **sh** does not.
    c. Which option automatically exports variables when they are assigned a value?
    d. Which option might you use to debug a shell script?
    e. What character would you use to separate three commands on the same line?
    f. What character starts a comment in a shell script?
    g. Which directory does the ~jim notation reference?
    h. What is the output of the **echo 'My shell is $SHELL'** command?
    i. How do you enter "In-line Editing mode" (assume you have EDITOR=vi)?
    j. Explain the difference between the **. myscript** and **myscript** commands.

72. leave
    a. What function does the **leave** command perform?
    b. When might you use it?

73. let
    a. What is an alternate syntax format to the **let** command?
    b. Write the command to display the result of 1024 times 100, for the **ksh** and the **csh**.
    c. Which command does the **let** command replace?

74. line
    a. What does the **line** command do?

75. listusers
    a. What information is displayed by the **listusers** command?

76. ln
    a. What is the purpose of the **ln** command?
    b. Write the command to link file to filex.

77. login
    a. Why does the **login** command exist?

78. logname
    a. What does the **logname** command do?

79. lp
    a. What is the function of the **lp** command?
    b. How do you suppress the request information message?
    c. Write the command to print the /etc/group file to the lzr printer.

80. lpq
    a. Why might you use the **lpq** command?
    b. Write the command to display information about the imagen printer.

81. lpr
    a. What does the **lpr** command do?
    b. Write the command to print file1 to the hplj printer.

82. lprm
    a. What is the purpose of the **lprm** command?
    b. Write the command to remove print job 1362.

83. lpstat
    a. What information does the **lpstat** command provide?
    b. Write the command to display all information about the spooler.

84. ls
    a. What function does the **ls** command serve?
    b. Write the command to display all files in all subdirectories.
    c. Write the command to display nonprintable characters.
    d. Explain each position of the file modes.
    e. Write the command to allow other users to execute a program as though they owned the program.

85. machid
    a. What is the function of the **machid** commands?
    b. Why might you use these commands?

86. mail/mailx
    a. What is the purpose of the **mail/mailx** command?
    b. What is the format of the **mailx** command to send mail to a user?
    c. What is the format for reading mail from your mbox file?
    d. How do you edit a message you are sending?
    e. When reading mail how do you display the next page of header lines?
    f. When reading mail how do you delete messages 20 through 34?
    g. Write the internal command to send message 5 to the dp printer.

87. man
    a. What does the **man** command do?
    b. Type the command to display information about the **vi** command.
    c. On BSD what command searches man headers for matching words?

88. mesg
    a. What is the purpose of the **mesg** command?
    b. Write the command to prevent messages?

89. mkdir
    a. Explain why or why not the command **mkdir src/cmd src** works.
    b. What does the SV -m option do?
    c. On SV how can you make the command in a. work?

90. more/page
    a. What is the purpose of the **more** command?
    b. When might you want to use **more**?
    c. Write the internal command to find the first occurrence of "string" in the current text of **more** input.
    d. How do you scroll down the input?

91. mv
    a. What are three uses of the **mv** command?
    b. What two commands does the **mv** command use to perform its job?
    c. Write the command to rename ../file1 to src/myprog.c.

92. newgrp
    a. Describe the function of the **newgrp** command.
    b. Why would you use the **newgrp** command?

93. news
    a. What does the **news** command do?
    b. How do you force all news items to be displayed?

94. nice
    a. What does the **nice** command change?
    b. What are some uses of the **nice** command?

95. nl
    a. Describe the function of the **nl** command.
    b. Write the command to number lines in increments of 10 and zero fill five spaces.

96. nohup
    a. What is the purpose of the **nohup** command?
    b. Where is the output of the nohup command written?
    c. Can you redirect the output to a file?

97. od
    a. What formats of output data does the **od** command support?
    b. What does the -c option do?
    c. How do you display data in hexidecimal format?

98. pack/unpack
    a. What does the **pack** command do?
    b. How do you reverse the effects of the **pack** command?
    c. Do you have to specify the .z extension when unpacking a file?

99. Parameters/Variables/Arrays
    a. What is the shell "environment"?
    b. How can you set positional parameters from within a shell?
    c. Which variable contains the return status of the last command?
    d. Write the command for assigning "Subscript 100" to the 100th position of the array ex_array.
    e. What is the syntax for referencing positional parameter 23?
    f. Name one situation where using an array would be better than using variables.

100. passwd
    a. Explain password aging.
    b. What are the requirements of SV passwords?
    c. What are two reasons for using the **passwd** command?

101. paste
    a. List the functions of the **paste** command.
    b. Write the command to display the standard input as 5 columns.

102. pcat
    a. When do you use the **pcat** command?
    b. What extension must the file have to use **pcat**?

103. pg
    a. List some of the features of the **pg** command?
    b. What does the -n option do?
    c. Write the command to reset the **pg** prompt to "Press CR to continue."
    d. How do you search for a string from the command line?
    e. What types of commands are supported by **pg**?
    f. How do you search backward from the **pg** prompt?
    g. Write the command to skip forward 3 files.

104. popd
    a. Explain what the **popd** command does.
    b. Which command must be used before **popd** can be used?

105. pr
    a. What is the function of the **pr** command?
    b. Write the command to convert input tabs to 4 spaces.
    c. How do you double space the output using **pr**?
    d. Write the command to change the header to "My test pr command."
    e. Generate a listing of files in 3-column output using just **pr**.

106. print
    a. Explain the two different **print** commands.
    b. Write two versions of **print** that do not print a new-line.
    c. How do you save the output of the print command to the history file?

107. printenv
    a. What is the purpose of the **printenv** command?
    b. When might you use the **printenv** command?

108. Programming Constructs - Shell Programming
    a. List the steps you follow to create a shell script.
    b. Change the **lg** shell script to display the information field of the /etc/passwd file. The information field is field 5.
    c. List the three types of looping conventions supported by the shells.
    d. Why use the **case** statement instead of the **if** statement?
    e. What are the benefits of using functions?

109. ps
    a. What is the function of the **ps** command?
    b. If the output is greater than 132 columns, what option do you use to display it correctly?
    c. How do display information about all processes on SV and or BSD?

110. pushd
    a. Explain the directory stack.
    b. Why do you use the directory stack?

111. pwd
    a. What does the **pwd** command do?
    b. Why use the **pwd** command?

112. read/readonly/set $<
    a. What does the **read** command do with the input it reads?
    b. What does the **readonly** command do that **read** does not?
    c. If you have a command **read X Y** and your input is "INPUT HERE," what is the value of Y?

113. Regular Expressions (REs)
    a. Explain a regular expression.
    b. Why are they so valuable?
    c. Write the regular expression to match multiple occurrences of the same character.
    d. How do you reference regular expressions that have been grouped?

114. rm
    a. What function does the **rm** command perform?
    b. Why might you use the **rm** command?
    c. How do you remove directories with the **rm** command?

115. rmdir
    a. What condition must the directory be in before **rmdir** will work?
    b. Explain the -p option on SV **rmdir**.

116. script
    a. What is the purpose of the **script** command?
    b. What file does the **script** file create by default?

117. sed
    a. What is **sed**?
    b. Explain how it basically works.
    c. Write the command to subsitute all occurrences of "system" with "computer."
    d. Which commands allow you to insert text?
    e. Write a **sed** command to display the first 100 lines of a file.

118. set/unset
    a. What functions does the **ksh** and **sh set** command provide?
    b. What does the **csh set** command do?
    c. How do you display the local environment using **set** (**ksh** and **sh**)?
    d. Write the **csh** command to set TERM to vt100.

119. setenv/unsetenv
    a. What does the **setenv** command do with variables?
    b. Can a child process change the environment of a parent process?
    c. How could you pass a variable's value to a parent shell?

120. shift
    a. What does the **shift** command do?
    b. Write the command to shift all but one parameters (hint: use the eval command).

121. sleep
    a. Explain the sleep command.

122. sort
    a. What type of files does the **sort** command sort?
    b. Write the command to sort on field 5.
    c. How do you sort numbers?
    d. Can **sort** reverse the order of the sort?
    e. How do you remove duplicate adjacent lines?

123. spell
    a. How can you use your own list of correctly spelled words?
    b. What are the shortcomings of the **spell** command?

124. split
    a. Explain how the **split** command functions.
    b. Write the command to split myfile into 10 line files whose names begin with mysplit.

125. stop
    a. What does the **stop** command do?
    b. What state must a process be in before you can use the **stop** command on the process?
    c. How does the **ksh** implement the **stop** command?

126. stty
    a. What is the purpose of the **stty** command?
    b. Write the SV and BSD command to display all settings.
    c. Write the command to change your erase key to Ctrl-B and your kill key to Ctrl-K.
    d. What happens if you type the **stty -echo** command?
    e. What happens if you disable isig (stty -isig)?
    f. What is the default EOF character for **stty** (UNIX)?

127. su
    a. What is the function of the **su** command?
    b. Explain what happens if you don't use the - option.
    c. Why might you use the **su** command?

128. sum
    a. List some uses of the **sum** command.

129. suspend
    a. What is the purpose of the **suspend** command?
    b. What is the format of the **suspend** command?
    c. How do you restart a suspended job?

130. tabs
    a. Explain what the **tabs** command does.
    b. What is meant by "canned" tab formats?
    c. Why must **tabs** know your terminal type?

131. tail
    a. What is the purpose of the **tail** command?
    b. What does the -f option cause **tail** to do?
    c. Write the command to skip the first 25 lines of myfile.

132. talk
    a. Describe the talk command.
    b. Write the command to talk to john at utvax.

133. tar
    a. What is the function of the **tar** command?
    b. Why might you use the **tar** command?
    c. Write the command to tar your current directory to /dev/rmt/0m.

134. tee
    a. What is the function of the **tee** command?
    b. What does the -a option do?
    c. What is the name of the output files from **tee**?

135. test
    a. Explain the function of the **test** command.
    b. Write the command to compare two numbers, $X and $Y.
    c. Write the command to find out if a file is readable and writable.

136. time/timex
    a. What is the function of the **timex** command?

137. time/times
    a. What is the purpose of the **times** command?
    b. Explain how the **time** command functions.

138. touch
    a. What is the function of the **touch** command?
    b. What is the difference between the SV and BSD **touch** commands?
    c. Write the command (SV) to create a file with the date of Nov 11, 1989 at 12:00 noon.

139. tput
    a. What does the **tput** command do?
    b. Write the command to clear your terminal screen.
    c. Write the command to move your cursor to position 10 20 on your screen.

140. tr
    a. What is the function of the **tr** command?
    b. Write the command to translate all numbers to dashes.

141. trap/onintr
    a. Explain what the **trap** command does.
    b. Write the **trap** command to rm all files in /tmp with names ending in .$$ if you press the Delete key (Signal 2).

142. true
    a. What is the function of the **true** command?
    b. What is the return code of the **true** command?

143. tset
    a. Explain what the **tset** command does.
    b. Write the command using **tset** to set your erase key to Ctrl-H and set your terminal type to vt100 if you're on a tty with a baud rate higher than or equal to 9600.
    c. What file might you place the **tset** command in for regular use?

144. tty
    a. What information does the **tty** command display?
    b. When might you use the **tty** command?

145. typeset
    a. What does the **typeset** command allow you to do with shell variables?
    b. Write the command to display a variable with 5 positions and zero filled.
    c. What does the -l option cause **typeset** to do?

146. ulimit/limit
    a. What does **ulimit** limit?
    b. What does the **limit** command limit?
    c. How can you as a user reset your ulimit to a higher value?

147. umask
    a. Explain the function of the **umask** command.
    b. What does the **umask 023** command force for permissions on newly created files?

148. uname
    a. What information is displayed by the **uname** command?

149. uniq
    a. What function is performed by the **uniq** command?
    b. How do you display only uniq lines?
    c. Write the command to compare only columns 20 and higher.

150. users
    a. What information is displayed by the **users** command?
    b. Why would you use the **users** command?

151. vi/view
    a. What is **vi**?
    b. What must you do before using **vi**?
    c. How do you start **vi** and have it position you at a certain string in the file?
    d. What are the six operators used by **vi**?
    e. How do you recover a file using **vi**?
    f. What are the modes within **vi**?
    g. Explain what a register is in **vi**.
    h. Describe how to copy text between two buffers.
    i. How do you move to the top line of the buffer?
    j. Write the command to filter the entire buffer through the **sort -u** command.
    k. How do you mark an address and how do you reference the marked address?
    l. Create a string macro to write the current file to disk and edit the next file in the argument list.

152. wait
    a. What function does the **wait** command serve?
    b. When do you use it?

153. wall
    a. What does the **wall** command do?
    b. When is it often used?
    c. Who usually uses the **wall** command?

154. wc
    a. What units are displayed by the **wc** command?
    b. Write the command to count the number of characters in all files beginning with a "f."

155. whatis
    a. Why would you use the **whatis** command?
    b. Is the information displayed always decisive and helpful?

156. whence/type
    a. Explain the function of the **whence** command.
    b. Why do we need the **whence** command?

157. whereis
    a. What information does the **whereis** command display?
    b. How do you force **whereis** to only search for man page entries?
    c. Can you change the directories **whereis** searches? If so, how?

158. which
    a. What is the function of the **which** command?
    b. What does the **which** command display if the command is not located?

159. who
    a. What is the function of the **who** command?
    b. How many options does the BSD version support?
    c. What does the -u option do?
    d. How can you display the number of users currently on the system?

160. whoami
    a. What information does the **whoami** command display?

161. write
    a. What does the **write** command do?
    b. How do you terminate the **write** command to send the message?

162
163
164 What is fc'H?
with b=-4 from Class X to BASIC, 13-*** including
C++ as STREAMS, it logs as follows:-
Lo → 22 OCTOBER 2015 10:16 AM, CORTEX
next traversal: morden physics

Appendix C
MS-DOS TO UNIX COMMAND CROSS REFERENCE

The following table lists MS-DOS and UNIX commands that perform similar functions. It is advisable to look up the module for each of the UNIX commands listed to gain a complete understanding of the command.

MS_DOS Command	*UNIX Command*	*Description*
CHDIR (CD)	**cd**	Change directory
CLS	**tput clear, clear**	Clear terminal screen
COMMAND	**sh, ksh, csh**	Command interpretors (user shells)
COMP	**diff, cmp, comm**	Compare files
COPY	**cp, cpio, tar, dd**	Copy a file or group of files
	cat	Concatenate files
DATE	**date**	Display and change system date. Only super-user can change the date on UNIX.
DIR	**ls**	List directory contents
EDLIN	**ed, ex**	Line editor
ERASE, DEL	**rm**	Remove/delete files
FIND	**grep, egrep, fgrep, nawk, sed**	Search for text in files
MKDIR (MD)	**mkdir**	Make a directory
MORE	**pg**	Display files one screen at a time
PRINT	**lp**	Print files in background(spooling)
RENAME	**mv**	Move/rename files
RMDIR (RD)	**rmdir**	Remove a directory
SET	**set**	Set value of shell variable
SORT	**sort**	Sort contents of files
TIME	**date**	Display and change system time
TYPE	**cat, tail, head, nawk, sed**	Display contents of a file

Appendix D
VMS TO UNIX COMMAND CROSS REFERENCE

The following table lists VAX VMS and UNIX commands that perform similar functions. It is advisable to look up the module for each of the UNIX commands listed to gain a complete understanding of the command.

VAX/VMS Command	UNIX Command	Description
SET DEFAULT	**cd**	Change directory
SHOW DEFAULT	**pwd**	Display present working directory.
CLS	**tput clear, cls**	Clear terminal screen
CREATE	**touch**	Create an empty file.
	sh, ksh	Command interpretors (user shells)
	diff, cmp, diff3, comm	Compare files
COPY	**cp, cpio, tar, dd**	Copy a file or group of files
	cat	Concatenate files
SHOW TIME	**date**	Display and change system date
DIR	**ls**	List directory contents
	ed, ex	Line editor
EDT	**vi**	Screen editor
DELETE, PURGE	**rm**	Remove/delete files
	grep, egrep, fgrep, nawk, sed	Search for text in files
	mkdir	Make a directory
	pg, more, less	Display files one screen at a time
PRINT	**lp**	Print files in background(spooling)
SHOW QUEUE	**lpstat**	Display print queue.
RENAME	**mv**	Move/rename files
DELETE/DIR	**rmdir**	Remove a directory
	sort	Sort contents of files
TYPE	**cat, tail, head, nawk, sed**	Display contents of a file
Symbols	**alias x="command"**	Setting up symbols (aliases)

Appendix E
FILE FORMATS

This appendix provides detailed information on the formats of files used by the *user* utilities.

PASSWD FILE

The /etc/passwd file contains information about each user. It is an ASCII file which you can read. The fields are colon separated. If the password field is left empty, no passwd is requested at login. If the login_sh field is empty, the /bin/sh is used for the login shell.

The format is:

```
user_name:password:user_id:group_id:comment(username):home_directory:
  login_sh
```

GROUP FILE

The /etc/group file contains information about all groups on the system and who has access to groups other than their primary group. The /etc/group file is an ASCII file which you can read. The fields are colon separated. Each group appears on a line by itself. If the passwd field is empty, no passwd is requested when changing groups.

The format is:

```
group_name:group_passwd:group_id:user_names
```

The user_names field is a list of valid users separated by commas.

OTHER FILES

For additional file formats refer to your system's Section 5 man pages.

Appendix F
ASCII COLLATING SEQUENCE

The ASCII collating sequence is used by most, if not all, UNIX Systems. This sequence is sometimes referred to as the machine collating sequence. Computers use codes to represent human alphabets and symbols. It is referred to as a collating sequence because the code is an increasing set of numeric values or codes.

The ASCII code uses numeric values from 0 to 255. The standard character set, the control characters, the alphabet, numbers, and symbols are represented by values of 0 through 127. Some computers may use different collating sequence codes. For example IBM uses EBCDIC. The ASCII code and EBCDIC codes are not compatible. If you tried to sort a file of EBCDIC codes with an ASCII systems' sort you would get bizarre output that would not be sorted correctly.

The following table is a complete listing of the ASCII code. The table lists the binary, octal, decimal, and hexidecimal equivalents for each ASCII code. The KEY column is the keyboard key the codes represent. The ASCII column is the ASCII defined label. The NAME column defines what the character is referred to by most users.

BINARY	OCTAL	DECIMAL	HEX	KEY	ASCII	NAME
0	0	0	0	^@	NUL	Null
1	1	1	1	^A	SOH	
10	2	2	2	^B	STX	Start of text
11	3	3	3	^C	ETX	End of text
100	4	4	4	^D	EOF	End of file
101	5	5	5	^E	ENQ	
110	6	6	6	^F	ACK	Acknowledge
111	7	7	7	^G	BEL	Bell
1000	10	8	8	^H	BS	Backspace
1001	11	9	9	^I	HT	Hard Tab
1010	12	10	A	^J	NL	New Line
1011	13	11	B	^K	VT	Vertical Tab
1100	14	12	C	^L	FF	Form Feed
1101	15	13	D	^M	CR	Carriage Return
1110	16	14	E	^N	SO	
1111	17	15	F	^O	SI	
10000	20	16	10	^P	DLE	
10001	21	17	11	^Q	DC1	
10010	22	18	12	^R	DC2	
10011	23	19	13	^S	DC3	
10100	24	20	14	^T	DC4	
10101	25	21	15	^U	NAK	No Acknowledge-ment

BINARY	OCTAL	DECIMAL	HEX	KEY	ASCII	NAME
10110	26	22	16	^V	SYN	Synchronize
10111	27	23	17	^W	ETB	
11000	30	24	18	^X	CAN	Cancel
11001	31	25	19	^Y	EM	
11010	32	26	1A	^Z	SUB	
11011	33	27	1B	^[	ESC	Escape
11100	34	28	1C	^\	FS	
11101	35	29	1D	^]	GS	
11110	36	30	1E	^^	RS	Sometimes ^~
11111	37	31	1F	^_	US	
100000	40	32	20		SP	Spacebar
100001	41	33	21	!		Exclamation Mark (Ball Bat)
100010	42	34	22	"		Double Quote
100011	43	35	23	#		Number Sign
100100	44	36	24	$		Dollar Sign
100101	45	37	25	%		Percent Sign
100110	46	38	26	&		Ampersand
100111	47	39	27	'		Single Quote (grave accent)
101000	50	40	28	(		Left Parenthesis
101001	51	41	29	)		Right Parenthesis
101010	52	42	2A	*		Asterisk
101011	53	43	2B	+		Plus Sign
101100	54	44	2C	,		Comma
101101	55	45	2D	-		Hyphen, Minus
101110	56	46	2E	.		Period
101111	57	47	2F	/		Slash
110000	60	48	30	0		Zero
110001	61	49	31	1		One
110010	62	50	32	2		Two
110011	63	51	33	3		Three
110100	64	52	34	4		Four
110101	65	53	35	5		Five
110110	66	54	36	6		Six
110111	67	55	37	7		Seven
111000	70	56	38	8		Eight
111001	71	57	39	9		Nine
111010	72	58	3A	:		Colon
111011	73	59	3B	;		Semicolon
111100	74	60	3C	<		Less Than Sign, Left Arrow
111101	75	61	3D	=		Equal Sign
111110	76	62	3E	>		Greater Than Sign, Right Arrow
111111	77	63	3F	?		Question Mark
1000000	100	64	40	@		At Sign

BINARY	OCTAL	DECIMAL	HEX	KEY	ASCII	NAME
1000001	101	65	41	A		
1000010	102	66	42	B		
1000011	103	67	43	C		
1000100	104	68	44	D		
1000101	105	69	45	E		
1000110	106	70	46	F		
1000111	107	71	47	G		
1001000	110	72	48	H		
1001001	111	73	49	I		
1010101	125	85	55	U		
1001010	112	74	4A	J		
1001011	113	75	4B	K		
1001100	114	76	4C	L		
1001101	115	77	4D	M		
1001110	116	78	4E	N		
1001111	117	79	4F	O		
1010000	120	80	50	P		
1010001	121	81	51	Q		
1010010	122	82	52	R		
1010011	123	83	53	S		
1010100	124	84	54	T		
1010110	126	86	56	V		
1010111	127	87	57	W		
1011000	130	88	58	X		
1011001	131	89	59	Y		
1011010	132	90	5A	Z		
1011011	133	91	5B	[		Left Bracket
1011100	134	92	5C	\		Backslash
1011101	135	93	5D	]		Right Bracket
1011110	136	94	5E	^		Caret
1011111	137	95	5F	_		Underscore
1100000	140	96	60	`		Acute accent, Back quote
1100001	141	97	61	a		
1100010	142	98	62	b		
1100011	143	99	63	c		
1100100	144	100	64	d		
1100101	145	101	65	e		
1100110	146	102	66	f		
1100111	147	103	67	g		
1101000	150	104	68	h		
1101001	151	105	69	i		
1101010	152	106	6A	j		
1101011	153	107	6B	k		
1101100	154	108	6C	l		
1101101	155	109	6D	m		
1101110	156	110	6E	n		

BINARY	OCTAL	DECIMAL	HEX	KEY	ASCII	NAME
1101111	157	111	6F	o		
1110000	160	112	70	p		
1110001	161	113	71	q		
1110010	162	114	72	r		
1110011	163	115	73	s		
1110100	164	116	74	t		
1110101	165	117	75	u		
1110110	166	118	76	v		
1110111	167	119	77	w		
1111000	170	120	78	x		
1111001	171	121	79	y		
1111010	172	122	7A	z		
1111011	173	123	7B	{		Left Brace
1111100	174	124	7C	|		Vertical Bar, Pipe
1111101	175	125	7D	}		Right Brace
1111110	176	126	7E	~		Tilde
1111111	177	127	7F	^?		Delete

EXTENDED (8-BIT) ASCII

BINARY	OCTAL	DECIMAL	HEX	KEY	ASCII	NAME
10000000	200	128	80			
10000001	201	129	81			
10000010	202	130	82			
10000011	203	131	83			
10000100	204	132	84			
10000101	205	133	85			
10000110	206	134	86			
10000111	207	135	87			
10001000	210	136	88			
10001001	211	137	89			
10001010	212	138	8A			
10001100	214	140	8C			
10001101	215	141	8D			
10001110	216	142	8E			
10001111	217	143	8F			
10010000	220	144	90			
10010001	221	145	91			
10010010	222	146	92			
10010011	223	147	93			
10010100	224	148	94			
10010101	225	149	95			
10010110	226	150	96			
10010111	227	151	97			
10011000	230	152	98			

BINARY	OCTAL	DECIMAL	HEX	KEY	ASCII	NAME
10011001	231	153	99			
10011010	232	154	9A			
10011011	233	155	9B			
10011100	234	156	9C			
10011101	235	157	9D			
10011110	236	158	9E			
10011111	237	159	9F			
10100000	240	160	A0			
10100001	241	161	A1			
10100010	242	162	A2			
10100011	243	163	A3			
10100100	244	164	A4			
10100101	245	165	A5			
10100110	246	166	A6			
10100111	247	167	A7			
10101000	250	168	A8			
10101001	251	169	A9			
10101010	252	170	AA			
10101011	253	171	AB			
10101100	254	172	AC			
10101101	255	173	AD			
10101110	256	174	AE			
10101111	257	175	AF			
10110000	260	176	B0			
10110001	261	177	B1			
10110010	262	178	B2			
10110011	263	179	B3			
10110100	264	180	B4			
10110101	265	181	B5			
10110110	266	182	B6			
10110111	267	183	B7			
10111000	270	184	B8			
10111001	271	185	B9			
10111010	272	186	BA			
10111011	273	187	BB			
10111100	274	188	BC			
10111101	275	189	BD			
10111110	276	190	BE			
10111111	277	191	BF			
11000000	300	192	C0			
11000001	301	193	C1			
11000010	302	194	C2			
11000011	303	195	C3			
11000100	304	196	C4			
11000101	305	197	C5			
11000110	306	198	C6			
11000111	307	199	C7			

BINARY	OCTAL	DECIMAL	HEX	KEY	ASCII	NAME
11001000	310	200	C8			
11001001	311	201	C9			
11001010	312	202	CA			
11001011	313	203	CB			
11001100	314	204	CC			
11001101	315	205	CD			
11001110	316	206	CE			
11001111	317	207	CF			
11010000	320	208	D0			
11010001	321	209	D1			
11010010	322	210	D2			
11010011	323	211	D3			
11010100	324	212	D4			
11010101	325	213	D5			
11010110	326	214	D6			
11010111	327	215	D7			
11011000	330	216	D8			
11011001	331	217	D9			
11011010	332	218	DA			
11011011	333	219	DB			
11011100	334	220	DC			
11011101	335	221	DD/D			
11011110	336	222	DE			
11011111	337	223	DF			
11100000	340	224	E0			
11100001	341	225	E1			
11100010	342	226	E2			
11100011	343	227	E3			
11100100	344	228	E4			
11100101	345	229	E5			
11100110	346	230	E6			
11100111	347	231	E7			
11101000	350	232	E8			
11101001	351	233	E9			
11101010	352	234	EA			
11101011	353	235	EB			
11101100	354	236	EC			
11101101	355	237	ED			
11101110	356	238	EE			
11101111	357	239	EF			
11110000	360	240	F0			
11110001	361	241	F1			
11110010	362	242	F2			
11110011	363	243	F3			
11110100	364	244	F4			
11110101	365	245	F5			
11110110	366	246	F6			

BINARY	OCTAL	DECIMAL	HEX	KEY	ASCII	NAME
11111000	370	248	F8			
11111001	371	249	F9			
11111010	372	250	FA			
11111011	373	251	FB			
11111100	374	252	FC			
11111101	375	253	FD			
11111110	376	254	FE			
11111111	377	255	FF			

Appendix G
vi QUICK REFERENCE

This appendix provides quick reference information on how to use the **vi** editor.

Substitutions Substitutions are handled by the **ex** portion of the **vi** editor. Refer to Module 43 for further discussion about **ex** and substituting.

Tags Tags are discussed in Module 43, which discusses the **ex** editor.

vi Command Table This table provides a quick reference to the motion and four most used operators combined with the motions.

Unit of Text	*Delete*	*Change*	*Copy/(yank)*	*Filter*	*Indent*
Character					
Current	x	r	yl	NA	NA
Previous	X	hr	yh	NA	NA
Next	lx	lr	lyl	NA	NA
Word					
Beginning next	dw	cw	yw	NA	NA
End current	de	ce	ye	NA	NA
Beginning previous	db	cb	yb	NA	NA
Up to next ip\(dg	dW	cW	yW	NA	NA
Previous ip	dB	cB	yB	NA	NA
Line					
Current	dd	cc or S	yy or Y	!!	>>
Beginning of	d0	c0	y0	NA	NA
End of	d$ or D	c$ or C	y$	NA	NA
Previous	dk	ck	yk	!k	>k
Next	dj	cj	yj	!j	>j
Forward to c	df*c*	cf*c*	yf*c*	NA	NA
Back to c	dF*c*	cF*c*	yF*c*	NA	NA
Sen/Para/Sec					
Previous sentence	d(	c(	y(	!(	>(
Next sentence	d)	c)	y)	)!	>)
Previous paragraph	d{	c{	y{	!{	>{
Next paragraph	d}	c}	y}	!}	>}
Previous section	d[[	c[[	y[[	![[	>[[
Next section	d]]	c]]	y]]	!]]	>]]

Unit of Text	*Delete*	*Change*	*Copy/(yank)*	*Filter*	*Indent*
Screen					
Beginning of	dH	cH	yH	!H	>H
Middle of	dM	cM	yM	!M	>M
End of	dL	cL	yL	!L	>L
Searches					
Previous pat	d?*pat*	c?*pat*	y?*pat*	!?*pat*	>?*pat*
Next pat	d/*pat*	c/*pat*	y/*pat*	!/*pat*	>/*pat*
Marks					
Line marked m	d'*m*	c'*m*	y'*m*	!'*m*	>'*m*
Character marked m	d‘*m*	c‘*m*	y‘*m*	!‘*m*	>‘*m*
Buffer					
Beginning of	d1G	c1G	y1G	!1G	>1G
End of	dG	cG	yG	!G	>G
Line number x	d*x*G	c*x*G	y*x*G	!*x*G	>*x*G

Customizing Options The **ex/vi** editor has many options that you can set to customize how the editor functions. Please refer to Module 43 for a discription of all supported options.

Index

! escape to shell, **ed**, 156
! escape to shell, **ex**, 180
! escape to shell, **pg**, 419
! ignore unsaved modification, **ex**, 172
! logical NOT, **find**, 237
! logical NOT, **let/@**, 298
! logical NOT, **nawk**, 46
! negation, **let/@**, 298
! negation, **nawk**, 40
! negation, **test**, 543
! variable, last background process, 399
!! repeat last escape to shell, **ex**, 172
!= not equal to, arithmetic, 40
!= not equal to, arithmetic, **let/@**, 298
!= not equal to, string, 40
!~ no existence of string, @, 299
!~ nonmatch string operator, 39
" reference register, **vi**, 606
"..." constants, 41
"..." quoting, 288
alternate buffer, **vi**, 598
alternate filename, **ex**, 172
nawk comment, 41
sed script comment, 482
shell comment, 286
shortest left substring, **ksh**, 405
variable, number of parameters, 399
longest left substring, **ksh**, 405
$ default primary **ksh** prompt, 401
$ last line, **ed**, 156
$ last line, **ex**, 171
$ last line, **sed**, 477
$ last mail message, 345
$ RE, end of line, 461
$ variable, process ID of script, 399
$! process ID of last background job, 614
$() command substitution, 286, 287
$? return code variable, 100
% buffer addresses 1,$, **ex**, 171
% current buffer, **vi**, 598
% current filename, **ed**, 156
% current filename, **ex**, 172
% last replacement RE, 155
% match {}, (), **ex**, 189
% match {}, (), **vi**, 603
% modulus, **expr**, 204
% modulus, **let/@**, 298
% remainder, **expr**, 204
% shortest right substring, **ksh**, 405
%% current job, 59, 221
%% longest right substring, **ksh**, 405
%+ current job, 59, 221
%- previous job, 59, 221
%= modulus assignment, 45
%? match job name, 59, 221
%**job** job name, 59, 221, 270
& background operator, 285
& bitwise AND, **let/@**, 299
& last RE, 155, 466
& non-NULL or nonzero, **expr**, 205
& repeat last editor command, 156
& repeat last **ex** command, **vi**, 609
&& logical AND operator, **ksh**, 285, 435
&& logical AND, **let/@**, 299
&& logical AND, **nawk**, 40, 46
'' previous current line, **ex**, 171
'' previous current line, **vi**, 602
'...' absolute quoting, 288
'x jump to mark, **ex**, 171
'x jump to mark, **vi**, 602
(()) arithmetic evaluation, **ksh**, 297
() group operators, 40, 298, 544
() RE, group REs, 465
() subshell operators, 285
(), show matching, **ex**, 189
* all mail messages, 345
* multipy, **expr**, 204
* multipy, **let/@**, 298
* RE, zero or more, 461
* variable, all **ksh** parameters, 399
*= multiplication assignment, 45
+ addition, **expr**, 204
+ addition, **let/@**, 298
+ next line, **ex**, 171
+ RE, match one or more, 461, 465
++ increment operator, 46
+= addition assignment, 45
, buffer addresses 1,., **ed**, 156
, buffer addresses 1,., **ex**, 171
, repeat last F/T search, **vi**, 601
- previous line, **ex**, 171
- subtraction, **expr**, 205

- subtraction, **let/@**, 298
- unary minus, **let/@**, 298
- variable, current options, 399
-- decrement operator, 46
-= subtraction assignment, 45
. current directory, 11, 369
. current line, **ed**, 156
. current line, **ex**, 171
. current mail message, 345
. RE, single character, 461
. repeat last command, **vi**, 609
. source script, **ksh**, 290
.. parent directory, 11, 369
.= current line number, **ex**, 174, 185
.cshrc file, 260, 292, 352, 518, 626
.exrc file, 190
.exrc, ex init file, 187
.kshrc file, 260, 352, 401, 403
.login file, 260, 292, 295, 352, 571
.mailrc file, 353
.plan file, 239
.profile file, 260, 283, 295, 352, 358, 403, 507, 518, 571
.project file, 239
/ (root directory), 24
/ divide, **expr**, 204
/ division, **let/@**, 298
// RE, null or empty, repeat, 465, 601
// repeat search, **ex**, 171
// repeat search, **vi**, 601
/= division assignment, 45
/RE/ match RE operator, 39
/RE/ search for RE, **ed**, 153
/RE/ search for RE, **ex**, 171
/RE/ search for RE, **vi**, 601
: buffer addresses 1,$, **ed**, 156
: ex prompt, 188
: matching operator, **expr**, 203
: null command, **ksh**, 290
: temporary **ex** mode, **vi**, 603, 171
: true statement, 566
; command separator, **ex**, 172, 186
; empty statment, 52
; repeat last f/t search, **vi**, 601
; sequential execution operator, 284
< buffer addresses 1,., **ed**, 156
< deindent operator, **vi**, 592, 604, 605
< input redirection, 266
< less than, arithmetic, **let/@**, 298
< less than, arithmetic, **nawk**, 39
< less than, **expr**, 205
< less than, string, **nawk**, 39
< read input variable, **csh**, 400, 487
<&fd read from file descriptor, 266
<&p read from pipe, 267
<< bit shift left, **let/@**, 298
<<- here document, **ksh**, 266
<= less than or equal to, arithmetic, **let/@**, 298
<= less than or equal to, arithmetic, **nawk**, 40
<= less than or equal to, **expr**, 205
<= less than or equal to, string, **nawk**, 40
= assignment, **let/@**, 299
= current line number, **ed**, 156
= equal to, **expr**, 205
= pathname listing, **fc**, 215
= total lines, **ex**, 186
== equal to, arithmetic, **let/@**, 298
== equal to, arithmetic, **nawk**, 40
== equal to, string, **nawk**, 40
=~ existence of string, @, 299
> buffer addresses 1,$, **ed**, 156
> default secondary **ksh** prompt, 401
> greater than, arithmetic, **let/@**, 298
> greater than, arithmetic, **nawk**, 39
> greater than, **expr**, 205
> greater than, string, **nawk**, 39
> indent operator, **vi**, 592, 604, 605
> output redirection, 14, 49, 266
>! force overwriting of file, **csh**, 266
>& write standard error, **csh**, 267
>&fd write to file descriptor, 266
>= greater than or equal to, arithmetic, **let/@**, 298
>= greater than or equal to, arithmetic, **nawk**, 40
>= greater than or equal to, **expr**, 205
>= greater than or equal to, string, **nawk**, 40
>> append output redirection, 49, 266
>> bit shift right, **let/@**, 298
>| force overwriting of file, **ksh**, 266
? RE, match zero or more, 461, 465
? variable, existence test, **csh**, 399, 405
? variable, last exit status, **ksh**, 399
?: conditional expression, 46
?? repeat search, **ex**, 171
?? repeat search, **vi**, 601
?RE? search backward for RE, **ex**, 171
?RE? search backward for RE, **vi**, 601

@ alias expansion, **fc**, 215
@ arithmetic evaluation, **csh**, 297
@ at the baud rate of, **test**, 570
@ invoke **vi** macro, 608
@ variable, all **ksh** parameters, 399
@@ repeat **vi** macro, 608
[test command, 541
[...] RE, character class, 461, 462
[^...] RE, not in character class, 463
\ escape special character meaning, 288, 461, 462, 467
\ filename completion, **fc**, 215
\ pathname completion, **fc**, 215
\(...\) RE, group REs, 464
0dd ASCII character in octal , 467
b backspace, 467
c literl character c, 467
E turn off \U or \L, **ex**, 184, 466
e turn off \U or \L, **ex**, 184, 466
f form feed, 467
l convert to lowercase, **ex**, 184, 466
L convert word to lowercase, **ex**, 184, 466
n carriage return, 467
r new-line, 467
t tab, 467
u convert to uppercase, **ex**, 184, 466
U convert word to uppercase, **ex**, 184, 466
\{\} RE, occurrences of pattern, 464
^ bitwise XOR, **let/@**, 299
^ first mail message, 345
^ RE, beginning of line, 461
_ variable, last positional parameter, 399
'...' command substitution, 286
'**x** jump to mark, **ex**, 171
'**x** jump to mark, **vi**, 601
" previous current line, **ex**, 171
" previous current line, **vi**, 602
{} group statement, **nawk**, 51
{} group statement, **sed**, 482
{} grouping keywords, **ksh**, 285
{} show matching, **ex**, 189
| bitwise OR, **let/@**, 299
| command separator, **ex**, 172, 186
| non-NULL or non-zero, **expr**, 205
| pipe creating operator, **ksh/csh**, 284
| print output to command, **nawk**, 50
| RE, either or, 461, 465
|& write standard error on pipe, **csh**, 267
|| logical OR operator, **ksh**, 285, 435
|| logical OR, **let/@**, 299
|| logical OR, **nawk**, 40, 46
~ bitwise negation, **let/@**, 298
~ directory substitution, 72, 286
~ invert character case, **vi**, 605, 609
~ last replacement string, **ex**, 184, 466
~ match string operator, 39
~+ absolute path, PATH, 286, 491

A
abbreviations, **vi**, 608
absolute addressing, **pg**, 417
absolute file permission modes, 80
absolute path, 71
access control, tty, 368
access permission, crontab, 113
access to history file, 211, 261, 289
access to **vi**, 170
access, system, 310
accessing registers, **vi**, 598, 606
account defaults, 5
account setup, 5
account, login to new, 518
accounting records, 547
accounting variable, 401
action, 36
add directory to stack, 454
address process is waiting on, 449
addresse units, **vi**, 592, 597
addressing, **ed**, 150
addressing, **ex**, 171
addressing, **sed**, 477
adjusting screen, **vi**, 602
aging, password, 409
alias, 29
alias expansion, **fc**, 215
alias group, mail, 340, 346
alias, filenames, 307
alias, preset, 31
alias, tracked, 31 256, 485
aliases, display value of, 622
all mail messages, 345
all values of array, **ksh**, 404
allow messages to tty, 367
alter CPU priority, 385
alternate buffer, **vi**, 598
alternate flow control, 435
anchoring regular expressions, 463
AND operator, binary, **test**, 543

append command, **ed**, 152
append command, **ex**, 175
append command, **sed**, 479
append command, **vi**, 604
append mode, **ed**, 152
append mode, **ex**, 175
append mode, **vi**, 604
append next line, **sed**, 480
append to hold space, **sed**, 480
append to registers, **vi**, 598
appending lines, **diff**, 136
apropos command, 33
archive files, 105
arctangent function, **atan2**, 47
ARGC variable, 41
argument list, **ex**, 186
arguments, keyword, 485
arguments, parse, 247
arguments, process, 247
arguments, resetting shell, 485
arguments, setting shell, 485
argv variable, 400
ARGV variable, 41
arithmetic assignment operators, 45, 299
arithmetic base variable, 576
arithmetic calculations, 297
arithmetic expressions, 297
arithmetic operators, 204, 297
arithmetic variables, 297
arithmetic, integer, 300
array element, **delete** function, 47
array elements, **ksh**, 403
array elements, **nawk**, 43
array subscripts, **ksh**, 403-405
array subscripts, **nawk**, 44
array, **ksh**, 403
array, **nawk**, 43
arrays, deleting elements, 45
arrays, looping through, 44
arrays, splitting strings into, 45
ascending sort, 500
ASCII, 13, 67
ASCII codes, 599
ASCII collating sequence, 665
ASCII to EBCDIC, 127
assign attributes to variables, 575
assign input to variables, 458
assign values to variables, 575
assignment operators, arithmetic, 45, 299
assumptions, 5
atan2, arctangent function, 47
attached process, 270
attributes, file, 79, 236
attributes, terminal, 555, 569
attributes, variable, 400, 575
autoindent option, **ex**, 187
autoindent, backtab over, **vi**, 604
autoload command, 401
automatic displaying, 529
automatic editor write(save), 187
automatically export variables, 485
automatically set variables, 399
autoprint option, **ex**, 187
autowrite option, **ex**, 187
awk command, 35
awk, see **nawk**

B
background command, **bg**, 59
background editor, **ex**, 180
background job, wait to finish, 614
background operator, 285
background process, 22, 23, 270, 507
backslash quoting, 288
backspace character, 8
backup file, create, 104
banner command, 55
base number system, 300
basename command, 57
baud rate, 510
beautify option, **ex**, 187
BEGIN pattern in **nawk**, 37
beginner mode, **ex**, 188
bg command, 59
bidirectional conversation, 633
blank interpretation, 287
block count, 521
block input operations, 550
block size, input, 127
block size, output, 127
block special file, 26, 333
boolean operators, **nawk**, 40
boot, last system, 629
bottom of file, display, 529
Bourne shell, see **ksh**
BPI rating, 536
branch command, **sed**, 479
break long lines, 245

break loop statement, 51
break out of loop, 439
break signal, 511
break statement, 439
British spelling, 502
broadcast message, 617
BSD, 2
buffer position, marking **ed**, 153
buffer position, marking **ex**, 176
buffer position, marking **vi**, 602
buffer units, **vi**, 592, 597
buffer, alternate, **vi**, 598
buffer, current, **vi**, 598
buffer, movement, **vi**, 602
buffer, moving around, **vi**, 592
buffers **vi**, 598
build directory, 370
build shell scripts, 432
building block, 21
built-in variables, 41
byte compare files, **cmp**, 88
byte dump, 392
byte size, 510
byte swapping, **cbyte swapping**, **dd**, 127

C
C programming language, 23
cal command, 61
calculate checksum, 521
calculations, arithmetic, 297
calendar command, **cal** , 61
calendar command, **calendar** , 63
calling other users, 531
cancel print request, 65
canned tab stops, 526
capnames, terminal, 555
case statement, 435
cat command, 67
cat packed file, 414
catch signals, 562
cd command, 71
cdpath variable, 402
CDPATH variable, 72, 400
change accounts, 518
change command, **ed**, 152
change command, **ex**, 175
change command, **sed**, 479
change directories, 422, 454
change directory owner, 85
change directory permissions, 79
change directory, **cd**, 71
change environment, 164, 484, 490, 519
change file owner, 85
change file permissions, 79
change files group ID, **chgrp**, 76
change group ID, 519
change groups, 380
change operator, **vi**, 592, 604
change passwd file information field, **chfn**, 74, 408
change password , 12, 408
change shell options, 486
change text, **vi**, 605
change user ID, 519
change viewing environment, 418
changing lines, **diff**, 137
changing password, 12, 408
changing text, **ex**, 175
character count, 618
character movement, **vi**, 599
character pitch, 315
character search, **vi**, 601
character special files, 26, 333
character units, **vi**, 592, 597
character variables, 41
characters unused, **vi**, 598
characters, complement, 599
characters, converting to/from lowercase, 126
characters, converting to/from uppercase, 127
characters, create large, 55
characters, delete, 599
characters, formatting, 145
characters, lowercase, 7
characters, map input, 511
characters, map input, **vi**, 607
characters, nonprintable , 68, 153
characters, replace, 560
characters, translate, 599
characters, uppercase, 7
chdir command, 71
check for mail variable, 401
check for syntax errors, 485
check mail variable, 402
checking, spell, 501
checksum, calculate, 521
chfn command, 74
chmod command, 79
chown command, 85

class, printer, 326
clear command, 87
clear screen, 87, 556
clobber option, **ksh**, 485
close file descriptor, 195
close standard input, 267
close standard output, 267
closing files, **nawk**, 51
closing pipes, **nawk**, 51
cmp command, 88
codes, ASCII, 599
col command, 90
colon command, 290, 566
columns, 35, 121
COLUMNS variable, 214, 331, 372, 400
columns, tty, 514
combine files horizontally, 412
combining files, 15
comm command, 93
command, 12
command being executed, 448
command execution, 99
command format, **ex**, 174
command format, **ksh**, 96
command history file, 260
command interpreter, 12
command line arguments, **ARGV**, 41
command line variable assignment, 42
command line, **nawk**, 42
command mode, **ed**, 149
command mode, mail, 345
command mode, **vi**, 596, 599
command processor, 280
command separator, **ex**, 172, 186
command separator, shell, 96
command substitution, 286
command syntax (rules), 96
command, **alias**, 29
command, display path to, 622, 624, 626
command, searching for, 99
command, time to execute, 546, 549
commands, 96
commands, built-in, 100
commands, common, 13
commands, display previous, 262
commands, external, 100, 289
commands, internal, 100, 289
commands, keyword, 98
commands, mail, 346
commands, motion, **vi**, 592
commands, pipeline, 98
commands, simple, 98
comments, **ksh**, 286
comments, **nawk**, 41
comments, **sed**, 482
common concepts variables, 10
communicate with another user, 531
communicate with users, 18
compare ASCII files, **comm**, 93
compare ASCII files, **diff**, 134
compare directory contents, 139
compare expressions, 204
compare files, **cmp**, 88
compare files, **diff**, 134
comparison operators, **expr**, 205
comparison operators, **nawk**, 39
complement characters, 599
completion, filename, 215
completion, pathname, 215
compound patterns, 40
compress data, 395
compressed print, 315
concatenate files, 67
concatenation regular expressions, 467
conditional commands, 433
conditional expressions, 46, 541
conditional operators, 205
conditional operators, **tset**, 570
cons of **vi**, pros and, 593
constants, **nawk**, 41
context splitting, 118
continue statement, 51, 440
control characters, discard, 187
control characters, display, 595
control characters, insert, **vi**, 604, 607
control characters, tty, 513, 570
control code sequences, 513, 570
control CPU usage, 385
control modes, tty, 510, 570
control resources, 576
control shell actions, 484
control system access, 310
control terminal attributes, 558, 570
control tty access, 368
control, job, 221, 270, 508
control-flow statements, 434, 435
control-flow statments, **nawk**, 51
conversation, bidirectional, 531, 633

convert data (EBCDIC/ASCII), 126
convert number to string, 53
convert spaces to tab, 90
convert string to number, 53
convert to lowercase, 511
convert to uppercase, 511
convert type variables, 42
convert variable to lowercase, 576
convert variable to uppercase, 576
convert, carriage return, 510
convert, input characters, 511
copy command, **ed**, 155
copy command, **ex**, 175
copy directory structures, 109
copy file descriptor, 195
copy files and directories, 105
copy mail message, 347
copy of terminal session, 474
copying files, 15, 68, 103, 109
copying text, **ex**, 175
core file, maximum size, 576
correct previous command, 214
correct spelling, 501
correction keys, default, 8
cos, cosine function, 47
cosine function, **cos**, 47
count characters, 618
count lines, 618
count words, 618
cp command, 103
cpi, characters per inch, 315
cpio command, 105
CPU priority, alter, 385
CPU seconds consumed, 550
CPU time allowed, maximum, 576
CPU utilization, 449
create alias, 256
create archive, 105, 109, 534
create backup file, 104
create calendar, 61
create directories, **cpio**, 107
create file, 10, 14, 157, 592
create large characters, 55
create multiple files, 505, 592
create password, 408
create shell scripts, 432, 433
create spelling list, 501
create zero byte file, 552
creating directories, 14
creating files, 14, 69
crontab access allowed, 113
crontab access denied, 113
crontab command, 111
crontab file format, 111
crypt command, 115
csh, 2
csh shell, 280
csplit command, 118
Ctrl-Q, stop output, 511
Ctrl-S, start output, 511
Ctrl-Z, suspend character, 186, 523
cumulative system time, 549
cumulative time of process, 449
cumulative user time, 549
current buffer, **vi**, 598
current directory, 11, 369, 456
current field, **NF**, **nawk**, 41
current filename, **FILENAME**, **nawk**, 41
current hostname, 263
current mail message, 345
current record, **FNR**, **nawk**, 41
current record, **NR**, **nawk**, 41
cursor movement, 557
customizing options, **ex**, 186
cut command, 121
cut operator, **vi**, 592, 604, 606
cutting text, **ex**, 176
cutting text, **vi**, 606
cwd variable, 400

D

data conversion, 126
data encryption, 115, 148, 168, 589
data security, 115
data segment, maximum size, 576
data space, unshared, 550
data structure, array, 44, 403
data type, 223
data, in columns, 35, 121
data, in rows, 35
data, lines, 35
data, sort, 497
data, typeless, 35
date command, 123
date format, 124
date, display, 123
date, set, 123
dd command, 12

DEAD variable, 354
debug shell scripts, 485, 486
dec mode, tty, 515
decipher, 116
decrease CPU priority, 385
decrement operators, 46
decrypt data, 115, 149, 589
default variable assignment, 403
define variable as function, 576
define variables, 484, 490
defining terminal type, 593
definitions, 635
definitions, input mode motion, **vi**, 608
definitions, **vi**, 596
deindent command, **ex**, 177
deindent operator, **vi**, 592, 604, 605
deindenting text, **ex**, 177
deindenting text, **vi**, 605
delay execution, 495
delete characters, 599, 604, 605
delete command, **ed**, 152
delete command, **ex**, 176
delete command, **sed**, 479
delete command, **vi**, 605
delete directories, 469, 472
delete duplicate lines, 586
delete files, 569
delete key, 13
delete links, 469
delete operator, **vi**, 592, 604
delete print jobs, 323
delete, array element function, 47
deleted lines, **vi**, 595, 605
deleting array elements, 45
deleting files, 15
deleting lines, **diff**, 136
deleting text, **ex**, 176
deleting text, **vi**, 605
delimeter, see separator
delimeters, paragraph, **ex**, 188
delimeters, sections, **ex**, 189
delimit options, 247
deny messages to tty, 367
descend directory structure, 234
descending sort, 500
design of **vi**, 592
destination printer, 314
destroy a file, 67
detached process, 270
determining terminal type, 593
/dev/null, garbage can, 291
/dev/tty terminal device, 573
device driver, set options, 509
device drivers, 26
device independence, UNIX, 24
devices, tape, 536
df command, 130
diagnostic message, **ex**, 189
dial-up password, 309
dictionary order sort, 498
differences between directories, 139
differences between files, 88, 93, 134
direct terminal I/O, 573
directories, compare, 139
directories, creating, 14
directories, display, 329
directories, limitations, 24
directories, link together, 305
directories, remove, 469, 472
directories, renaming, 15, 377
directories, search for, 234
directory, 24, 333, 369
directory contents, list, 329
directory modes, mask, 582
directory modes, set, 582
directory option, **ex**, 187
directory owner, **chown**, 85
directory path, **CDPATH**, 72
directory permissions, 333
directory permissions, test, 541
directory stack, 142, 421, 454
directory substitution, 286
directory, current, 11, 369
directory, differences, 139
directory, home, 11
directory, make new, 369
directory, max files, 370
directory, new, 369
directory, parent, 11, 369
directory, present, 11, 71, 456
dirname command, 141
dirs command, 142
disable filename generation, 485
disable filename wildcards, 485
disable job monitor, 484
disable option, 486
disable parity checking, 511
disable printer, 327

discard control characters, 187
disconnect login, 309
disk free space, 130
disk space used, 143
disk space, reduce required, 396
display bottom of file, 529
display command, **ed**, 145
display command, **ex**, 174
display date, 123
display directories, 329
display duplicate lines, 586
display end of file, 529
display environment, 431
display file automatically, 529
display file contents, 13, 67
display file size, 619
display filenames, 329
display full name, 239
display idle time, 239, 628
display last lines, 529
display line numbers, 68, 387, 425
display logged in users, 588, 628
display login name, 239
display login shell, 239
display login time, 239
display multiple files, 68
display office location, 239
display operating system information, 584
display packed file, 414
display path to command, 622, 624, 626
display pathname only, 141
display phone number, 239
display previous commands, 262
display print queue, 325
display printer forms, 326
display shell environment, 431, 484, 490
display status, **vi**, 603
display tabs, 188
display terminal device, 573
display terminal name, 239
display time, 123
display top of file, 258, 387
display uniq lines, 586
display variables, 145, 429, 484, 490
display, the **vi**, 595
displaying control charaters, **vi**, 595
divert input to file, 539
divide files, 118, 506
do loop statement, 51
documentation, 28
dot command, 290
dot directory, 369
dot dot directory, 369
double digit parameters, 404
double qoutes, 288
dream, 21
dumb terminals, **ex**, 188
dumb terminals, **vi**, 595
duplicate lines, delete, 586
duplicate lines, remove, 497
duplicate lines, report, 586
dynamic file growth, 24
D_SPELL variables, 503

E
EBCDIC to ASCII, 127
echo characters, tty, 512, 570
echo command, 145
echo variable, 402
ed editor, 16, 148
edcompatible option, **ex**, 187
edit alternate buffer/file, **vi**, 603
edit command, **ed**, 152
edit history file, 211, 289
edit mail message, 338, 342
editing commands, **diff**, 137
editing commands, **ed**, 152
editing commands, **ex**, 173
editing commands, **sed**, 478
editing history file, **csh**, 217
editing, text, 16
EDITOR variable, 289, 354, 400
editor, full screen, 589
editor, line, 148, 168
editor, stream, 476
editor, visual, 16
effective user name, 632
egrep command, 160
elapsed execution time, 546, 549
electronic post office, 338
elements, array, 43
elif statment, 434
elite pitch, 315
else statment, 434
emacs editing mode, **fc**, 216
empty file, create, 552
empty regular expressions, 465
empty statment, ;, 52

emulation, software, 594
emulation, terminals, 594
enable job monitor, 484
enable parity checking, 511
enable printer, 327
encrypt data, 115, 148, 168, 589
end of file, display, 529
END pattern in **nawk**, 38
end-of-file, EOF, 196
enter **ex** mode from **vi**, 603
env command, 164
ENV variable, 283, 401
ENVIRON, **nawk** environment variable, 41
environment, 10, 164, 201, 398, 484, 490, 519
environment space, 201, 490
environment variable, **ENVIRON**, **nawk**, 41
environment variables, 398, 484, 490
environment, change viewing, 418
environment, display, 431, 484, 490
environment, initialized, 309
environment, place variables in, 201, 398, 484, 490
environment, print, 431, 484, 490
eof character, tty, 513
EOF, end-of-file, 196
eol character, tty, 514
equality check file, 89
erase character, 8, 512, 514, 570, 604
erase word character, tty, 516
erorrbells option, **ex**, 187
ERRNO variable, 399
error messages, **ex**, 189
error messages, **vi**, 595
error variable, 576
errors, spelling, 501
escape character, 8, 604
escape from insert mode, **vi**, 604
escape sequences, 145
escape sequences, regular expressions, 466
escape sequences, remove, 90
escape to shell, 156, 180
establish terminal session, 308
/etc/login, 283
/etc/motd file, 519
/etc/profile, 283, 518
/etc/ttytype, 569
eval command, 166
evaluate arithmetic expressions, 297
evaluate conditional expressions, 541
evaluate expressions, 203
evaluate numbers, 541
evaluate strings, 541
event address, 449
event disgnators, **csh** history, 217
events, history commands, 211
ex editor, 16, 168
ex macros, 185
ex mode, **vi**, 603
ex read from shell command, 154, 180
ex tags, 184
ex write to shell command, 155, 180
ex/last_line mode, **vi**, 596
exception search command, **ed**, 155
exception search command, **ex**, 182
exchange hold and pattern space, 481
exec command, 194
execute command, time to, 546, 549
execute permission, 83, 333
execute previous command, 219
execute script, 290
execute, directory structure, 234
execution, command, 99
execution, delay, 495
exercises, 645
EXINIT variable, 187, 190
exit code, 100, 196, 208, 566
exit command, 196
exit from function, 442
exit statement, 51
exit status, 100, 196, 208, 566
exp, exponential function, 47
expand command, 199
expand command line, 166
expand filename, 232
expand input characters, 424
expand tabs to spaces, 199
expansion, alias, 215
expansion, filename, 230
expansion, variable, 404
exponential function, **exp**, 47
export command, 201
export variable, 576
exported variables, 201, 398
expr command, 203
expr?expr:expr, 46
expression comparison, 203
expression evaluation, 203
expression search, regular, 160, 181, 251

expression, parenthesized, 40, 298
expressions, arithmetic, 297
expressions, conditional, 541
expressions, file selecting, 237
extensions, filename, 229
extract archived files, 105, 109
extract files from archive, 534

F
false command, 208
fc command, 211
FCEDIT variable, 212, 401
features, shell, 280
fg command, 221
fi statement, 434
field separator, 35, 121
field separator, **FS**, **nawk**, 41
field separator, **sort**, 499
field variables, 42
field, key, 273
fields, **cut**, 121
fields, mail header, 342
fields, **nawk**, 35
FIFO, see pipes
fignore variable, 402
file attributes, 79
file command, 226
file command, **ed**, 152
file descriptor, close, 195, 267
file descriptor, copy, 195
file descriptor, open, 195
file descriptor, read from, 266
file descriptor, write to , 266
file descriptors, maximum number of, 576
file formats, 664
file locking, mandatory, 82, 83, 333
file modes, 333
file modes, mask, 582
file modes, set, 582
file naming rules, 230
file owner, **chown**, 85
file permission modes, **chmod**, 79
file permissions, 333
file permissions, test, 541
file recovery, **ex**, 168, 186
file recovery, **vi**, 594
file search, access time, 234
file search, current directory, 235
file search, current filesystem, 235
file search, file size , 235
file search, file type, 235
file search, filesystem type, 235
file search, group name, 235
file search, inode number, 235
file search, links, 235
file search, local filesystem, 235
file search, match filename, 235
file search, modify time, 235
file search, no group, 235
file search, no user, 235
file search, permissions, 235
file search, status time, 234
file statistics, 236
file system, 24
file system, characteristics, 24
file system, limitations, 24
file system, structure, 24
file type, 235, 332, 333
file, **.cshrc**, 260, 283, 518, 626
file, **.kshrc**, 260, 352, 401, 403
file, **.login**, 260, 283, 295, 352, 518, 571
file, **.mailrc**, 353
file, **.plan**, 239
file, **.profile**, 260, 283, 295, 352, 403, 507, 518, 571, 594
file, **.project**, 239
file, block special, 333
file, change group ID, **chgrp**, 76
file, character special, 333
file, command history, 260
file, compress, 395
file, create empty, 552
file, display bottom lines, 529
file, display end of, 529
file, display packed, 414
file, display top lines, 258, 387
file, dividing, 118
file, history, 260
file, maximum size, 576
file, ordinary, 333
file, pack, 395
file, peruse, 372, 416
file, scroll, 373
file, size, 330, 619
file, sort, 497
file, splitting, 118
file, unpack, 395
file, validate contents, 521

file, view, 372, 392, 416
filec variable, 232, 402
filename completion, 232
filename completion variable, 402, 403
filename completion, **fc**, 215
filename expansion, 230
filename extensions, 229
filename generation, 230
filename generation, disable, 485
filename wildcards, 230
FILENAME, current filename, **nawk**, 41
filenames, 229
filenames, display, 329
files, archive, 534
files, block special, 26
files, character special, 26
files, combine horizontally, 412
files, combining, 15
files, converting data, 126
files, copying, 15
files, create multiple, 505
files, creating, 14
files, cut columns out, 35, 121
files, cutting, 35, 121
files, deleting, 15
files, displaying, 13, 372, 416, 423
files, divide, 506
files, hard links, 305
files, hidden, 232, 329, 334
files, internal structure, 25
files, joining, 274
files, limitations, 26
files, link together, 305
files, listing, 13
files, locate, 234
files, merge, 425, 497
files, moving, 15, 377
files, moving text between, **vi**, 606
files, ordinary, 25
files, overwrite, 265
files, paste together, 412
files, problems, 26
files, remove, 469
files, renaming, 15, 377
files, search for, 234
files, selecting columns, 35, 121
files, soft links, 305
files, sort by time, 330
files, special, 26
files, split, 505
files, symbolic links, 305
fill variable with blanks, 575
fill variable with zeros, 576
filter command, **ed**, 154
filter command, **ex**, 177
filter operator, **vi**, 592, 604, 605
filter, terminate, 196
filtering text, **ex**, 177
filtering text, **sed**, 476
filtering text, **vi**, 592, 604, 605
filters, 27
filters, printer, 320
find command, 234
find match {}, (), **ex**, 189
find match {}, (), **vi**, 603
finger command, 239
first mail message, 345
flow control, 434
flow control, terminal, 511
flush character, tty, 515
flush input on signal, 512
fmt command, 242
FNR, current record in file, **nawk**, 41
fold command, 245
fold input lines, 424
fold long lines, 245
folders, mail, 348, 358
fonts, printer, 318
for statement, 52, 436
foreground a job, 221, 270
foreground process, 22, 23, 221, 270
format characters, 145
format files, 423
format text, 242
format within **vi**, 242
format, commands, 96
format, crontab file, 111
format, date, 124
format, directory listing, 330, 331
format, octal, 392
format, **printf**, 50
format, tab, 149, 526
format, time, 124, 549
formats, file, 664
formats, printer, 314, 320
forms, printer, 314, 320, 326
FPATH variable, 401
free space, disk, 130

FS, input field separator, **nawk**, 41
full disks, alleviating, 132
full name, display, 239
full pathnames, 10, 256
full regular expressions, 465
full screen editor, 589
full-screen editing, 170
function directory variable, 401
function, return from, 442
functions, arithmetic, 47
functions, built-in, 46
functions, **ksh**, 441
functions, **nawk**, 46
functions, string, 47
functions, user defined, 47

G
garbage can output, 268
garble terminal screen, 91
GECOS information, 74
generate calendar, 61
generate filenames, 229
generate reports, 35
generate spelling list, 504
getline funtion, **nawk**, 48
getopts command, 247
getting started, **vi**, 593
global exception search command, **ed**, 155
global exception search command, **ex**, 182
global RE print, 160, 223, 251
global search command, **ed**, 152
global search command, **ex**, 181
global substitute, **gsub** function, 47
global variables, **ksh**, 201, 398
global variables, **nawk**, 48
GMT, 123
goto address, **vi**, 602
GPID, group process ID, 277
grant write permission to tty, 367, 628, 624
grep command, 251
group access permission, 76
group file access permissions, 81
group ID, 264, 380
group ID, change, 519
group login, 380
group name, 264
group operators, (), 40, 298
group process ID, 277
group statement, {} , 51
grouping keywords, 285
grouping regular expressions, 464
groups command, 255
growing file, monitor, 529
gsub, global substitute function, 47

H
hang up phone, 510
hang-up, no, 390
hardpaths variable, 402
hardtabs, 526
hardtabs option, **ex**, 187
hardware requirements, 4
hardware tab spacing, 187, 525
hardware type, system, 584
hardware, identify, 336
hash command, 256
hashcheck command, 501
hashmake command, 501
head command, 258
head, page, 424
header line of man page, 620
headers, mail, 342
heterogeneous environment, 432
hexadecimal dump, 392
hidden files, 232, 329, 334
hierarchical file system, 24
histchar variable, 214, 262, 402
HISTFILE variable, 213, 261, 401
history command, 260
history file, 211, 261
history file editor, **FCEDIT**, 212
history variable, 213, 402
HISTSIZE variable, 213, 261, 401
hold print request, 314
hold space, **sed**, 480
HOME variable, 261, 353, 401, 402
home variable, 402
homogeneous environment, 432
horizontal movement, **vi**, 600
hostname command, 263
hostname, dipslay, 263
hostname, set, 263
hung process, 277
hung shell, 290
H_SPELL variables, 503

I
I/O, 9, 10, 265
I/O streams, 10, 265
I/O, direct terminal, 573
I/O, piping, 10, 265
I/O, read standard input, see reading
I/O, redirection, 10, 14, 194, 265
I/O, standard, 10, 265
I/O, standard error, 10, 265
I/O, standard input, 10, 265
I/O, standard output, 10, 265
id command, 264
identify hardware, 336
identify terminal type, 401, 569, 593
idle time, display, 239, 628
if statement, 52, 434
IFS variable, 287, 401
ignore carriage return, 511
ignore control characters, **sort**, 498
ignore leading blanks, **sort**, 499
ignore signals, 562
ignoreeof variable, 402
immune to hang-up, 390
implicit invocation, 284
in-line editor variable, 401
in-line history editing, 214, 289
increase CPU priority, 385
increment operators, 46
indent command, **ex**, 177
indent operator, **vi**, 592, 604, 605
indent text, 425, 605
indenting text, **ex**, 177
indenting text, **vi**, 592, 604, 605
indention width, **ex**, 189
index node, 25
index of next option, 248
index string function, **nawk**, 47
inform users of news, 383
information field, 74
information, pass to child process, 201
information, user, 239
ingorecase option, **ex**, 187
initial requirements, 4
initialize environment, 309
initialize terminal, 555, 569
inode number, 25
input block size, 127
input characters, map tty, 511
input characters, map, **vi**, 607
input control, **nawk**, 48
input field separator, **FS**, **nawk**, 41, 48
input mode motion definitions, **vi**, 608
input mode, **ed**, 149
input mode, **ex**, 175
input mode, **vi**, 596, 604
input modes, tty, 510
input record separator, **RS**, 42, 48
input, terminal, 268
Input/Output, see I/O
insert command, **ed**, 153
insert command, **ex**, 175
insert command, **sed**, 480
insert command, **vi**, 604
insert control characters, **vi**, 604, 607
insert mode, **ed**, 152
insert mode, escape from **vi**, 596, 604
insert mode, **ex**, 175
insert mode, **vi**, 596, 604
insert, repeat last, **vi**, 604
inserting text, **ex**, 175
inserting text, **vi**, 604
int, integer function, 47
integer arithmetic, 300
integer function, **int**, 47
integer variable, 576
interactive confirmation, copying files, 103
interactive full screen editor, 589
interactive mode, job, 221, 270
interactive renaming, **cpio**, 108
interactive, global command, **ed**, 153
interactive, substitution, **ex**, 182
interface to **ex**, 592
interface to shell, 170
interface to **vi**, 170
interface, hardware, 22
internal field separator, 401
internal, **mailx** variables, 353
interpret command line, 166
interpretation, variable, 404
interpretive programming language, 35
interpretive programming language, **ksh**, 280
interpretor, 432
interrupt character, tty, 514, 570
interrupt signal, 510
INTERRUPT, key, 13
invocation, shell, 283
items, news, 383
iteration commands, 436

J
job command, **bg**, 59
job control, 23, 221, 270, 508
job monitor, 485
jobs command, 270
join command, 273
join command, **ed**, 153
join command, **ex**, 178
join command, **vi**, 609
join files, 274
join lines, **ex**, 178
join lines, **vi**, 609
justification, right margin, 242

K
kernel, 1, 22
kernel stack size, 450
key field, 273
key, encryption, 115, 148, 152, 155, 168, 589
keyboard, 10
keys, backspace, 8
keys, correction, 8
keys, delete, 13, 595
keys, interrupt, 13, 595
keys, sort, 500
keystrokes, store multiple, 185
keyword commands, 98
keyword parameters, 400, 485
keyword variables, 400, 485
keywords, grouping, 285
kill character, 8, 512, 514, 570, 604
kill command, 276
kill **vi** session, 595
Korn shell, 280
ksh, 1, 2
ksh shell, 280
ksh, restricted, 284

L
label, **sed**, 481
large characters, create, 55
last argument, **ksh**, 404
last line, **vi**, 595
last lines, display, 529
last mail message, 345
last system boot, 629
leave command, 295
leave only filename, 406
leave only pathname, 406
left justify variable, 575
left margin justification, 525
left shift parameters, 492
length, page, 424
length, printer page, 314
length, string length function, 47
let command, 297
letters, see characters
limit, core file size, 576
limit, CPU time, 576
limit, data segment size, 576
limit, file descriptors, 576
limit, file size, 576
limit, hard, 576
limit, open files, 576
limit, soft, 576
limit, stack segment, 576
limit, virtual memory size, 576
limits, process, 450
limits, **vi**, 599
line command, 301
line count, 618
line editor, 168
line movement, **vi**, 600
line numbering **ex**, 188
line numbers, display, 68, 387, 425
line pitch, 315
line printer, 316
line units, **vi**, 592, 597
LINENO variable, 399
lines of data, 35
lines to scroll, **ex**, 189
LINES variable, 555
lines, fold long, 245
lines, join, **ex**, 178
lines, join, **vi**, 609
lines, see records also
link across file systems, 306
link files together, 305
links, hard, 305
links, remove, 469
links, soft, 305
links, symbolic, 305
lisp option, **ex**, 188
list command, **ed**, 153
list command, **sed**, 480
list directory contents, 329
list exported variables, 201
list groups, 255

list mail messages, 345
list option, **ex**, 188
list pathnames, 215
list user information, 239
LISTER variable, 355
listing files, 13
listusers command, 304
literals, 231
ln command, 305
local modes, tty, 512
local variables, 201, 398
locate a command, 622, 624, 626
locate files, 234
locate path to man page, 624
locate path to source code, 624
lock up terminal, preventing, 227
locking, mandatory file, 82, 83, 333
log, natural logarithm function, 47
logged in users, display, 588, 628
logging in, 7
logging out, 20
logical AND file operator, 237
logical AND operator, **ksh**, 285, 435
logical NOT file operator, 237
logical operators, **nawk**, 45
logical OR file operator, 237
logical OR operator, **ksh**, 285, 435
logical pages, 387
login account, 5
login command, 308
login disconnect, 309
login name, 312
login name, display, 239
login shell, display, 239
login time, display, 239
login time-out, 309
login to new account, 518
login to new group, 380
logname command, 312
LOGNAME variable, 312, 401, 628
logout command, 196, 197
looping commands, 436
looping through arrays, 44
loops, redirecting output, 268
lower CPU priority, 385
lowercase characters, 7
lp command, 313
LPDEST variable, 314
lpi, lines per inch, 315
lpq command, 318
lpr command, 320
lprm command, 323
lpsched command, 316
lpstat command, 325
ls command, 329

M
machid command, 336
machine type, 336
macros within macros, allow, **ex**, 189
macros, command mode, **vi**, 607
macros, **ex**, 185
macros, insert mode, **vi**, 607
magic characters, see regular expressions
magic file, 226
magic option, **ex**, 188
magnetic tape device, 536
mail alias group, 340, 346
mail checking variable, 401
mail command, 338
mail command mode, 345
mail commands, 346
mail folder, 348
mail headers, 342
mail message list, 345
mail recipients, 340
mail shell variables, 352
mail tilde escape commands, 342
MAIL variable, 353, 401, 402
mail, edit message, 338, 342
mail, organizing, 358
mail, read message, 338, 341, 344
mail, remote, 357
mail, review message, 338
mail, send message, 338, 341
mail, set variable, 350
mail, store message, 338
MAILCHECK variable, 401
MAILPATH variable, 401
MAILRC variable, 353
mailx command, 338
make directory, 369
man command, 363
man page, header line, 620
man page, locate path to, 624
man pages, 33
man pages, **man**, 363
mandatory file locking, 82, 83, 333

manipulation, string, 47, 206
manipulation, text, **vi**, 592, 603
manuals, on-line reference, 363
manuals, see documentation
manuals, see man pages
map command characters, **vi**, 607
map ESCAPE key, 485
map input characters, 511
map input characters, **vi**, 607
map uppercase to lowercase, 187
margin justification, left, 525
margin justification, right, 242
mark command, **ed**, 153
mark command, **ex**, 176
mark command, **vi**, 602
markdirs option, 485
marking text, **ed**, 153
marking text, **ex**, 176
marking text, **vi**, 602
mask directory modes, 582
mask file modes, 582
master directory, see root
match beginning of line, 463
match beginning of word, 463
match end of line, 463
match end of word, 463
match occurrences of pattern, 464
match pattern(s), **nawk**, 36
match string operator, **nawk**, 39
match {}, (), **ex**, 189
match {}, (), **vi**, 603
match, substring match function, 47
matched string length, **RLENGTH**, 41
matching operator, 203
max files in directory, 370
maximum CPU time allowed, 576
maximum number of file descriptors, 576
maximum real memory used, 550
maximum size core file, 576
maximum size data segment, 576
maximum size file, 576
maximum size stack segment, 576
maximum size virtual memory, 576
MBOX variable, 355
media type, physical, 536
memory address, 448
memory utilization, 449
memory, shared, 550
merge input files, 425, 497
mesg command, 367
mesg option, **ex**, 188
message to user, write, 633
message, broadcast, 617
messages, allow, **ex**, 188
metacharacters, **ex**, 188
metacharacters, **ksh**, 284
metacharacters, see regular expressions also
mkdir command, 369
mode, file, 333
modes, mask directory, 582
modes, mask file, 582
modes, see permission
modes, set directory, 582
modes, set file, 582
modes, **vi**, 596
modifiers, **csh** history, 217
modifiers, variable, 406
modify variable values, 406, 484, 490, 575
modularize, 21
module, type, 2
monitor growing file, 529
monitor, job, 485
more command, 372
MORE variable, 375
motd file, 519
motion commands, **vi**, 592, 599
move command, **ed**, 154
move command, **ex**, 176
move files, 15, 377
movement, **vi**, 599, 600
moving around buffer, **vi**, 592
moving text between files, **ex**, 177
moving text between files, **vi**, 606
moving text, **ex**, 176
moving text, **vi**, 606
MS-DOS to UNIX cross reference, 662
multicolumn output, 412, 423
multidimensional array, 44
multifile viewing, 372, 416
multiple files, combining, 69, 412
multiple files, copying, 69
multiple files, displaying, 68
multiple files, editing, 177
multitasking, 22
multiuser, 22
mv command, 377

N
name of system, 584
name, group, 255, 264
name, user, 264
named pipe, 333
named register, **ex**, 176
named registers, **vi**, 598
named variables, 400
naming rules, file, 230
natural logarithm function, **log**, 47
nawk, 35
nawk input control, 48
nawk output control, 49
nawk variables, 41
nawk, regular expressions, 38
nawk, select lines, 37
nawk, writing programs, 37
new environment, 164, 519
new-lines, visually diplay, 188
newgrp command, 380
news command, 383
news, for users, 383
NF, current field, 41
nice command, 385
nice value, 448
nightmare, 21
nl command, 387
nobeep variable, 403
noclobber option, **ksh**, 485
node name of system, 584
noglob variable, 403
nohup command, 390
nonmatch string operator, 39
nonomatch variable, 403
nonprintable characters, 68, 153, 480
notify to leave, 295
notify variable, 403
novice option, **ex**, 188
NR, current record, 41
null regular expressions, 465
null variable interpretation, 404
number command, **ed**, 154
number lines, 68, 387
number of arguments, **ARGC**, 41
number of array elements, **ksh**, 404
number option, **ex**, 188
number output lines, 425
number registers, scan, **vi**, 609
number registers, **vi**, 597
number to string, convert, 53
numbered variables, 398
numeric comparison tests, **test**, 543
numeric order sort, 498
numeric variables, **nawk**, 41, 42

O
objects, see units also
objects, **vi** text, 592, 597
octal dump, 392
office location, display, 239
OFMT, output format, **nawk**, 41
OFS, output field separator, **nawk**, 41
OLDPWD variable, 399
on-line reference manuals, 363
one dimensional array, **ksh**, 403
one dimensional array, **nawk**, 43
open buffer for insert, **vi**, 604
open file descriptor, 195
open mode, allow, **ex**, 188
open mode, **ex**, 170, 171
open option, **ex**, 188
operating system, 1
operating system information, 584
operator, background, 285
operator, change **vi**, 592, 604, 605
operator, cut **vi**, 592, 604, 606
operator, deindent **vi**, 592, 604, 605
operator, delete **vi**, 592, 604, 605
operator, filter **vi**, 592, 604, 605
operator, indent **vi**, 592, 604, 605
operator, logical AND, 285
operator, logical OR, 285
operator, matching, 203
operator, pipe creating, 284, 285
operator, sequential execution, 284
operator, yank **vi**, 592, 604, 606
operators, arithmetic assignment, 45, 299
operators, arithmetic, **expr**, 204
operators, arithmetic, **let/@**, 297
operators, **chmod**, 82
operators, decrement, 46
operators, grouping, 40, 298
operators, increment, 46
operators, logical, 45
operators, relational , 39
operators, string, **nawk**, 39
operators, string, **test**, 543
operators, subshell, 285

operators, test, 543
operators, text, **vi**, 592, 604
operators, tty baud rate, 570
OPTARG variable, 247, 398
optimize option, **ex**, 188
OPTIND variable, 247, 398
option, (no)clobber, **ksh**, 485
options, parse, 247
options, printer, 314
options, process, 247
options, resetting shell, 485
options, setting shell, 485
OR operator, binary, 544
ordinary file, 333
ordinary files, see files
organize mail, 358
ORS, output record separator, **nawk**, 41
other users file access permissions, 81
out of space, 231
output block size, 127
output control, **nawk**, 49
output field separator, **OFS**, **nawk**, 41
output format, **OFMT**, **nawk**, 41
output record separator, **ORS**, **nawk**, 41
output, terminal, 268
output, unbuffered, 67
overwrite existing files, 104
overwrite files, 265
owner file access permissions, 81

P

pack command, 395
pack data, 395
packed file, display, 414
page command, 372
page faults, number of, 550
page header, 424
page length, 424
page width, 425
PAGER variable, 354, 356
paragraph delimeters, **ex**, 188
paragraph movement, **vi**, 600
paragraph units, **vi**, 592, 597
paragraphs option, **ex**, 188
parameters, double digit, 404
parameters, keyword, 400
parameters, positional, 247, 398
parameters, see variables also
parent process ID, 448
parenthesized expression, 40, 298
parity checking, 511
parse positional parameters, 247
pass information to child process, 201
password, 7
password file information field, change, 74
password requirements, 410
password, aging, 409
password, changing, 12, 408
password, dial-up, 309
password, login, 308, 408
password, setting, 12, 408
password, **su**, 518
paste command, 412
paste files together, 412
pasting text, **ex**, 177
pasting text, **vi**, 606
PATH variable, 12, 256, 401, 403, 491
path variable, 403
pathname completion, **fc**, 215
pathname listing, **fc**, 215
pathname, display only, 141
pathnames, 10
pattern space, 477
pattern, match, 36
patterns, ranges, 40
pause after each page, 425
pcat command, 414
percentage of CPU utilization, 449
percentage of memory utilization, 449
permission strings, **chmod**, 82
permission to talk, 531
permission, group access, 76
permission, set execute, 83
permission, set mandatory file locking, 82, 83, 333
permission, set read, 82
permission, set set-group-ID, 82, 83
permission, set set-user-ID, 82, 83
permission, set sticky bit, 82, 83
permission, set write, 83
permission, special modes, 83
permissions file, 333
permissions, test, 541
perusal commands, **pg**, 417
peruse file, 372, 416
pg command, 416
philosophy, 21
phone number, display, 239

physical media type, 536
pica pitch, 315
PID, process ID, 277, 445
pipe creating operator, 284, 285
pipe mail message, 349
pipe output diversion, 539
pipe, named, 333
pipeline commands, 98
pipes, 26
piping, 10
pitch, printer character, 315
pitch, printer line, 315
popd command, 421
port identification, tty, 570
portability, 22
portability of **cpio** archives, 107
portable software, 23
positional parameters, 247, 398
positional parameters, setting, 484
positional parameters, sort, 486
post office, electronic, 338
PPID variable, 399
pr command, 423
precede with line numbers, 68, 387, 425
precise regular expressions, 464
predefined tab stops, 525
prefix, remove filename, 57, 406
prefix, remove pathname, 57, 406
present directory, 11, 71, 456
preserve editor buffer, **ex**, 186
preserved editor file, 168, 594
prevent filename expansion, 406
previous commands, display, 262
previous commands, see history file
print command, 428
print command, **ed**, 154
print command, **sed**, 480
print control characters, 68, 153
print control characters, **vi**, 595
print jobs, 313, 318
print non-ASCII characters, 68, 153, 480, 595
print queue, display, 325
print request **cancel**, 65
print shell environment, 431, 484, 490
print statement, **nawk**, 49
print text, 428
print to command, **nawk**, 50
printenv command, 431
printer class, 326
printer command, **cancel**, 65
printer fonts, 318
printer formats, 314, 320
printer forms, display, 326
printer queue, 313, 318
printer request priority, 315
printer status, 325
PRINTER variable, 318
printer, disable/enable, 327
printf format, 50
printf statement, **nawk**, 49
printf variable format, 50
priority, printer request, 315
priotiry, alter CPU, 385
private information, 116
procedure, shell, 281
process command line, 166
process control, 221, 270, 508
process execution time, 546, 549
process flags, 448, 450
process ID, 276, 445, 448
process kernel stack size, 450
process options, 247
process owner, 449
process priority, 448
process real memory size, 450
process signals, 563
process size, 449
process sleep time, 450
process sleeping, 495
process soft memory limit, 450
process status, 445, 448
process swap address, 449
process user ID, 449
process virtual memory size, 450
process, attached/detached, 270
process, hung, 277
process, in background, 22, 23, 270, 507
process, in foreground, 22, 23, 270
process, runaway, 277
process, stopped, 59, 507
process, suspended, 23, 59, 270, 495, 507, 523
process, terminate, 196
process, wait on background, 614
processor utilization, 448
program, shell, 281
programming examples, **nawk**, 53
programming language, C, 23
programming language, **ksh**, 280

programming language, **nawk**, 35
programming, shell, 432
prompt command, **ed**, 154
prompt option, **ex**, 188
prompt variable, 288
prompt variable, 288, 403
prompt, change **ed**, 148
prompt, **more**, 372
prompt, **page**, 372
prompt, primary shell, 401
prompt, secondary shell, 401
prompt, **select**, 401
prompting, shell, 288
pros of **vi**, 593
ps command, 445
PS1 variable, 401
PS2 variable, 401
PS3 variable, 401
PS4 variable, 401
pushd command, 454
put command, **ex**, 177
putting text, **ex**, 177
pwd, 11, 456
PWD variable, 399, 456

Q
Ctrl-Q, stop output, 511
QIC cartridge tape device, 536
query terminal database, 555
quick substitution, history commands, 212
quit command, **ed**, 154
quit command, **sed**, 480
quit signal, tty, 514
quoting, 288

R
random number function, **rand**, **nawk**, 47
RANDOM variable, 399
range patterns, **nawk**, 40
ranges, address, **ed**, 150
ranges, address, **ex**, 171
ranges, address, **sed**, 477
read command, 458, 487
read command, **ed**, 154
read command, **ex**, 180
read command, **sed**, 480
read from file descriptor, 266
read line, 301, 458
read mail message, 338, 341, 344
read permission, 82, 333
read-ahead feature, 8
read-only mode, **ex**, 188
read-only mode, **vi**, 589
readonly command, 458
readonly option, **ex**, 188
real memory used, 550
reassign positional parameters, 492
receiving mail, 19
recipients, mail, 340
record separator, **RS**, 42
records, 35
records, selecting, 37
recover **ex** file, 168, 186
recover **vi** file, 594
recursive, **nawk** functions, 48
recursively descend directories, 234
recursively, display directories, 329
redirecting output in loops, 268
redirection, see I/O
redo command, **ex**, 178
redraw option, **ex**, 188
reduce command search time, 256
reduce CPU priority, 385
reduce file size, 395
reduce muliple characters, 599
reduce redundant code, 441
reedit a file, **ex**, 178
reference manuals, on-line, 363
referencing tags, 185
referencing variables, 398, 404
register, macros, **vi**, 608
register, named, **ex**, 176
register, undo, **ex**, 176
registers, **vi**, 597
regular expressions, 38, 149, 161, 183, 251, 461-466
regular expressions, anchors, 463
regular expressions, concatenation, 467
regular expressions, empty, 465
regular expressions, escape sequences, 466
regular expressions, full, 465
regular expressions, grouping, 464
regular expressions, null, 465
regular expressions, precise, 464
regular expressions, simple, 461
regular expressions, single character, 462
regular expressions, special, 465
regular expressions, substitute, 465

relational operators , **nawk**, 39
relation database join, 273
relative addressing, **pg**, 417
relative path, 71
relative pathnames, 10
release of operating system, 584
remap option, **ex**, 189
reminder service, 295
reminder service, **calendar**, 63
remote mail, 357
remove blank lines, 586
remove control characters, **col**, 90
remove crontab file, 111
remove directories, 469
remove duplicate lines, 497, 586
remove empty lines, 68
remove escape sequences, **col**, 90
remove filename prefix, 57, 406
remove filename suffix, 141, 406
remove files, 469
remove leading blanks from variable, 575
remove links, 469
remove pathname prefix, 57, 406
remove print jobs, 323
remove top directory from stack, 421
remove variables, 484, 490
rename directories, 15, 377
rename files, 15, 377
repeat command line processing, 166
repeat last command, **vi**, 609
repeat last **ex** command, **vi**, 609
repeat last insert, **vi**, 604
repetitive tab stops, 526
replace argument list, **ex**, 179
replace characters, 560, 605
replace current process, 194
replace hold space, **sed**, 480
replace pattern space, **sed**, 480
replace text, **vi**, 605
reply to sender, 350
REPLY variable, 399
report duplicate lines, 586
report modifications, **ex**, 189
report option, **ex**, 189
report process status, 445
reprint character, tty, 515
request, printer, 314
request-ID, print, 65
requirements, password, 410
reserved variables, 399
reset arguments, shell , 485
reset **ex** options, 186
reset options, shell , 485
reset terminal, 556, 569
reset tty modes, 571
resident set size of process, 450
resident set size of text, 451
resource limits, 576
restrict sort fields, 498
restricted **ed**itor, 149
restricted **ksh**, 284
resume print request, 314
resume stopped job, 221
retrieve data, 35
retrieve text, **ex**, 178
retrieve text, **vi**, 597
return code, 100, 196, 208, 566
return statement, 442
return statement, **nawk**, 48
return status, 100, 196, 208, 566
reverse order sort, 498
review mail message, 338
rewind argument list, **ex**, 179
right justify variable, 575, 576
RLENGTH, matched string length, 41
rm command, 469
root directory, 24
rows of data, 35
rows, tty, 514
RS, record separator, 42
RSTART, start posistion of matched string, 42
rules, command syntax, 97
rules, file naming, 230
run level of system, 629
runaway process, 277

S
Ctrl-S, start output, 511
save and quit **ex**, 186
save mail message, 350
save text, **vi**, 597
save w/o checking, **ex**, 190
savehist variable, 213, 261, 403
scan number registers, **vi**, 609
screen, 10
screen adjusting, **vi**, 602
screen **clear**, 87, 556

screen control, **more**, 374
screen movement, **vi**, 600
screen units, **vi**, 592, 597
screen viewing, **more**, 374
screen, scroll, 416
screen, **vi**, 17
script command, 474
script of terminal session, 474
script, shell, 281, 432
scripts, **sed**, 476
scroll option, **ex**, 189
scroll screen, 373, 416
scrolling, **vi**, 602, 603
SCSI tape device, 536
search a file, 160, 223, 251
search command, **ed**, 152, 155
search command, **ex**, 181
search directories, 234
search directory permissions, 333
search for command, 99, 256
search for directory, 72, 234
search for files, 72, 234
search for pattern, 160, 223, 251
search for pattern, **pg**, 418
search for string, 160, 223, 251
search for word, 160, 223, 251
search man page header lines, 620
search, character, **vi**, 601
search, fast file, 223
search, man page header lines, 33
search, simple, 223
search, string, **vi**, 601
search, wrap around buffer, **ex**, 190
searche units, **vi**, 592, 597
SECONDS variable, 399
section movement, **vi**, 600
section units, **vi**, 592, 597
sections option, **ex**, 189
sections, man page, 365
security, 24
security, data, 115
sed command, 476
seed function, **srand**, 47
select lines, 37
select prompt, 401
selecting file, expressions, 237
send mail message, 338, 341
send messages to tty, 367
send signal to process, 276
send signals, device driver, 516
sending mail, 19
sentence movement, **vi**, 600
sentence units, **vi**, 592, 597
separate options, 247
separate words, shell, 287
separator, **ex** command, 172, 186
separator, field **FS**, **nawk**, 41
separator, field, **nawk**, 35, 121
separator, input field **FS**, **nawk**, 41, 48
separator, input field, **ksh**, 287, 401
separator, input record **RS**, **nawk**, 42, 48
separator, output field **OFS**, **nawk**, 41
separator, output record **ORS**, **nawk**, 41
separator, record **RS**, **nawk**, 42
separator, subscript **SUBSEP**, **nawk**, 42, 44
sequential execution operator, 284
session, killing **vi**, 595
set arguments, shell , 485
set command, 484
set date, 123
set directory permissions, 79, 582
set **ex** options, 186
set file permissions, 79, 582
set hostname, 263
set mail variables, 350
set options, **ex**, 186
set options, shell , 485
set read-only mode, **ex**, 188
set tab stops, 525
set terminal I/O options, 509, 569
set terminal type, 569, 593
set time, 123
set variable read-only, 576
set variables, 484, 490
set window size, **ex**, 190
set window size, **pg**, 419
set, **ex** option setting, 187
set-group-ID permission, 82, 333
set-user-ID permission, 82, 333
setenv command, 490
setting password, 12
sh command, 1, 2
SHACCT variable, 401
shared memory, 550
shell, 1, 27
shell arithmetic, 300
shell command separator, 96
shell command syntax, 98

shell environment, 201, 398, 484, 490, 519
shell environment, see environment
shell features, 280
shell option, **ex**, 189
shell procedure, 281
shell programming, 432
shell script, 281
SHELL variable, 356, 402
shell variable, 403
shell, see **csh**
shell, see **ksh**
shell, which version, 9
shift command, 492
shift posistional parameters, 492
shiftwidth option, **ex**, 189
shorthand command, see alias
shorthand for **ls**, 331
shorthand, file referencing, 230
showmatch option, **ex**, 189
showmode option, **ex**, 189
signal queue, 277
signal, break, 511
signal, hang-up, 390
signal, interrupt, 510
signals, 277
signals, catch, 562
signals, check for, 512
signals, ignore, 562
similarities, directory, 130
simple command syntax, 96
simple commands, 98
simple regular expressions, 461
simulaneous output redirection, 539
sin, sine function, 47
sine function, **sin**, 47
single character regular expressions, 462
single quotes, 288
size history file, 213, 261
size of file, 330, 619
size of process, 449
size window, **ex**, 174, 190
size window, **pg**, 419
skip backward, **pg**, 418
skip forward, **pg**, 418
sleep command, 495
slowopen option, **ex**, 189
software emulation, 594
software requirements, 4
sort ascending, 500
sort by time, files, 330
sort command, 497
sort descending, 500
sort keys, 500
sort positional parameters, 486
source code editing with tags, 168, 590
source code, locate path to, 624
source options, **ex**, 186
source script, **csh**, 290
source script, **ksh**, 290
spaces to tabs, convert, 199
spawn child process, 100
spawn new process, 445
special characters, filename generation, 229
special characters, **ksh**, 284
special characters, see regular expressions also
special modes permission, 83
special regular expressions, 465
special variables, 399
spell checking, 501
spell command, 501
spellin command, 501
spellout command, 501
split command, 505
split files, 505
split, string split function, **nawk**, 47
splitting a file, 118
splitting strings into arrays, **csh**, 407
splitting strings into arrays, **nawk**, 45
spool output to printer, 313, 318
sprintf, string format function, 47
sqrt, square root function, 47
square root function, **sqrt**, 47
squeeze occurrence characters, 599
srand, seed function, 47
stack segment, maximum size, 576
stack space, unshared, 550
stack, directory, 142
standard error, see I/O
standard I/O, 195
standard I/O, close, 195
standard I/O, copy, 195
standard I/O, open, 195
standard input, see I/O
standard output, see I/O
start posistion of matched string, **RSTART**, **nawk**, 42
start time of process, 449

state of job, 270
state of process, 270
statistics, file, 236
statistics, process execution, 546, 549
status line, **vi**, 595, 603
status of mail message, 342
status of process, 448, 450
status variable, 400
status, printer, 325
sticky bit permission, 82, 333
stop character, tty, 515
stop command, 507
stop editor, **ex**, 180, 186
stop process, 507
stopped process, 59, 507
storage, see directories
storage, see files
store mail message, 338
store multiple keystrokes, 185
store related information, see arrays
store text, **ex**, 176
store text, **vi**, 597
store values, 398, 484, 490, 575
store values, see arrays
stored tab stops, 526
stream editor, 476
Streams, see I/O
string comparison, 435
string comparison tests, 543
string format, **sprintf** function, 47
string manipulation, 47, 206
string operators, **nawk**, 39
string search, 160, 223, 251
string search, **vi**, 601
string **split** function, 47
string to number, convert, 53
string variables, **nawk**, 42
structure, file system, 24
structure, files, 25
structured programming, 433
structureless files, 24
stty command, 509
su command, 518
sub, substring search function, 47
subdirectories, search for files, 234
subscript separator, **SUBSEP**, 42, 44
subscripts, array, 44, 403-405
subsections, man page, 365
SUBSEP, subscript separator, 42, 44
subshell operators, 285
substitute **%** last replacement RE, 155
substitute **&** last RE, 155, 466
substitute command, **ed**, 154
substitute command, **ex**, 182
substitute command, **sed**, 480
substitute options, **ex**, 182
substitute regular expressions, 465
substitute, **gsub** function, 47
substitution, tilde, 286
substitution, variable, 404
substr, substring select function, 47
substring match, **match** function, 47
substring search, **sub** function, 47
substring select, **substr** function, 47
suffix, remove filename, 141, 406
sum command, 521
super-user, login as, 518
suspend character, tty, 515
suspend command, 523
suspended process, 23, 59, 270, 495, 407, 523
SV, 2
swapping bytes, **cpio**, 108
swapping bytes, **dd**, 127
swaps, number of, 550
switch between files, **ex**, 178
switch between programs, **ex**, 523
switch groups, 380
symbolic file permission modes, 82
symbolic labels, **ex**, 184
synchronize clock, 124
syntax errors, check for, 485
syntax rules, command, 97
system execute time, 546, 549
system hardware type, 584
system name, 584, 263
system node name, 584
system run level, 629
system time, cumulative, 549
S_SPELL variables, 503

T

tab format, 149, 526
tab right shiftwidth, **vi**, 604
tab spacing, hardware, 187, 525
tab stops, set, 525
table of contents, archive, 534
tabs command, 525
tabs to spaces, 199

tabs, visually display, 188
tabstop option, **ex**, 189
tag file, 184
taglength option, **ex**, 189
tags command, 184, 590
tags option, **ex**, 189
tags, referencing, 185, 590
tags, source code editing with, 168, 590
tail command, 529
talk command, 531
tape archive, 534
tape devices, 536
tar command, 534
task, 445
tee command, 539
term option, **ex**, 189
TERM variable, 170, 402, 403, 491, 555, 593
term variable, 403
termcap database, 555
TERMCAP variable, 402, 594
terminal device, display, 573
terminal garbage, preventing, 227
terminal I/O options, set, 509, 569
terminal input, 268
terminal keyboard, see keyboard
terminal name, display, 239
terminal output, 268
terminal reset, 556, 569
terminal screen, see screen
terminal session, copy of , 474
terminal session, establish, 308
terminal type, 11, 569, 593
terminal type variable, TERM, 401, 569, 593
terminal type, **ex**, 189, 190
terminal, flow control, 511
terminals emulation, 594
terminate filter, 196
terminate process, 196
terminfo database, 555
TERMINFO variable, 402, 594
terms, 635
terse mode, **ex**, 189
terse option, **ex**, 189
test command, 541
test conditional expressions, 541
test directory permissions, 541
test file permissions, 541
test number comparisons, 543
test string comparisons, 542
text manipulation, **vi**, 592, 603
text pattern search, 160, 223, 251
text, editing, see editing
text, formatting, 242
text, inserting, **ex**, 175
text, inserting, **vi**, 604
text, printing, 428
text, search for, 160, 223, 251
text, transposing, **vi**, 609
tilde escape commands, mail, 342
tilde substitution, 72, 286
time command, 546
time command, **csh**, 549
time format, 124
time to execute command, 546, 549
time variable, 403
time zone variable, 402
time, display, 123
time, set, 123
time-out, login, 309
timeout option, **ex**, 189
timeout, **ex**, 189
times command, **ksh**, 549
timex command, SV, 546
TMOUT variable, 402
too many arguments, 231
top of file, display, 258
touch command, 552
tput command, 555
tr command, 559
trace variable, 401
tracked alias, 31, 256, 485
transform command, **sed**, 481
transform data, 35
translate characters, 599
transport data, 110
transposing text, **vi**, 609
trap command, 562
tree structure, 24
true command, 566
tset command, 569
tty command, 573
tty control modes, 510
tty input modes, 510
tty port settings, 308
tty, control messages to, 367
tty, set options, 509
ttytype database, 659
ttytype option, **ex**, 190

type command, 622
type of data, 226
type of machine, 336
type variables, 398
type, also see version
type, data, 223
type, file, 332
type, terminal, 11, 569, 593
type, variable, 398
typeless data, 35
typeset command, 575
TZ variable, 402

U
ulimit command, 579
umask command, 582
unalias, 29
unbuffered output, 67
uncompress data, 395
undo command, **ed**, 155
undo command, **ex**, 178
undo command, **vi**, 609
undo register, **ex**, 176
undo register, **vi**, 597
unexpand command, 199
uniq command, 586
uniq duplicate lines, 497
units, addresses, **vi**, 592, 597
units, buffer, **vi**, 592, 597
units, characters, **vi**, 592, 597
units, lines, **vi**, 592, 597
units, paragraphs, **vi**, 592, 597
units, screen, **vi**, 592, 597
units, searches, **vi**, 592, 597
units, sections, **vi**, 592, 597
units, sentences, **vi**, 592, 597
units, words, **vi**, 592, 597
Universal time, 123
UNIX, 1
UNIX device independence, 24
UNIX, which version, 9
unnamed registers, **vi**, 597
unpack command, 395
unpack data, 395
unset command, 484
unset variable interpretation, 404
unset variables, 484, 490
unsetenv command, 490
unshared data space, 550
unshared stack space, 550
until loop, 439
unused characters, **vi**, 598
update dumb terminals, **ex**, 188
update file, 157
update file access time, 552
update file modification time, 552
uppercase characters, 7
uppercase to lowercase, 187
usage, disk, 143
user defined variables, 42
user execute time, 546, 549
user ID, 264
user ID of process, 449
user ID, change, 519
user information, 239
user name, 264
user time, cumulative, 549
user variable, 403
USER variable, 403, 491
user, talk to, 531
username, 312
username, **login**, 308
users command, 588
users file access permissions, 81
users, list, 304
users, write to all, 614
using mail folders, 358
utilization, CPU, 449
utilization, memory, 449

V
validate data, 35
validate file contents, 521
value of aliases, display, 622
value of variables, 400, 484, 490, 575
variable assignment, command line, 42
variable assignment, default, 403
variable expansion, 404
variable format, **printf**, 50
variable interpretation, 404
variable modifiers, 406
variable, **argv**, 400
variable, arithmetic base, 576
variable, **CDPATH**, 400
variable, **cdpath**, 402
variable, **COLUMNS**, 214, 331, 372, 400, 555
variable, convert to lowercase, 576

variable, convert to uppercase, 576
variable, **cwd**, 400, 456
variable, **DEAD**, 354
variable, define as function, 576
variable, **echo**, 402
variable, **EDITOR**, 289, 354, 400
variable, **ENV**, 283, 401
variable, **ERRNO**, 399
variable, error, 576
variable, **EXINIT**, 190
variable, export, 576
variable, **FCEDIT**, 212, 401
variable, **fignore**, 402
variable, **filec**, 232, 402
variable, fill with blanks, 575
variable, fill with zeros, 576
variable, **FPATH**, 401
variable, **hardpaths**, 402
variable, **histchar**, 214, 262, 402
variable, **HISTFILE**, 213, 261, 401
variable, **history**, 213, 402
variable, **HISTSIZE**, 213, 261, 401
variable, **HOME**, 261, 353, 401, 402
variable, **home**, 402
variable, **IFS**, 287, 401
variable, **ignoreeof**, 402
variable, integer, 576
variable, left justify, 575
variable, **LINENO**, 399
variable, **LISTER**, 355
variable, **LOGNAME**, 312, 401, 628
variable, **LPDEST**, 314
variable, **MAIL**, 353, 401
variable, **mail**, 402
variable, **MAILCHECK**, 401
variable, **MAILPATH**, 401
variable, **MAILRC**, 353
variable, **MBOX**, 355
variable, **MORE**, 375
variable, **nobeep**, 403
variable, **noglob**, 403
variable, **nonomatch**, 403
variable, **notify**, 403
variable, **OLDPWD**, 399
variable, **OPTARG**, 247, 398
variable, **OPTIND**, 247, 398
variable, **OPTION**, 247
variable, **PAGER**, 354, 356
variable, **PATH**, 12, 256, 401, 403, 491
variable, **path**, 403
variable, **PPID**, 399
variable, **PRINTER**, 318
variable, **prompt**, 288, 403
variable, **PS1**, 401
variable, **PS2**, 401
variable, **PS3**, 401
variable, **PS4**, 401
variable, **PWD**, 399
variable, **RANDOM**, 399
variable, remove leading blanks, 575
variable, **REPLY**, 399
variable, right justify, 575, 576
variable, **savehist**, 213, 261, 403
variable, **SECONDS**, 399
variable, set read-only, 576
variable, **SHACCT**, 401
variable, **SHELL**, 356, 402
variable, **shell**, 403
variable, **status**, 400
variable, **TERM**, 170, 402, 403, 416, 490, 555, 593
variable, **term**, 403
variable, **TERMCAP**, 402, 416, 594
variable, **TERMINFO**, 402, 416, 594
variable, **time**, 403
variable, **TMOUT**, 402
variable, **TZ**, 402
variable, **user**, 403
variable, **USER**, 403, 491
variable, **verbose**, 214, 403
variable, **VISUAL**, 356, 402
variables, arithmetic, 297
variables, assign attributes, 575
variables, assing input to, 458
variables, automatically export, 485
variables, built-in, 41
variables, character, 41
variables, common concepts, 10
variables, convert type, 42
variables, define, 484
variables, display, 429
variables, **D_SPELL**, 503
variables, environment, 398, 484, 490
variables, exported, 201, 398
variables, field, 42
variables, global, 201, 398
variables, **H_SPELL**, 503
variables, internal, **mailx**, 353

variables, keyword, 400
variables, local, 201, 398
variables, named, 400
variables, **nawk**, 41
variables, numbered, 398
variables, numeric, 41
variables, referencing, 398
variables, remove, 484
variables, reserved, 399
variables, see parameters also
variables, set, 484, 490
variables, set mail, 350
variables, special, 399
variables, **S_SPELL**, 503
variables, type, 398
variables, unset, 484, 490
variables, user defined, 42
variables, value of, 400, 484, 490, 575
verbose mode, **ksh**, 486
verbose variable, 214, 403
version 6 **cpio** format, 107
version control, **diff**, 137
version of **ex**, 186
version of operating system, 584
version of shell, 2, 9
version of UNIX, 2, 9
vertical movement, **vi**, 600
vi abbreviations, 608
vi buffers, 598
vi change operator, 592, 604
vi cut operator, 592, 604, 606
vi deindent operator, 592, 604, 605
vi delete operator, 592, 604, 605
vi display, 595
vi editor, 16, 589
vi filter operator, 592, 604, 605
vi indent operator, 592, 604, 605
vi limits, 599
vi macros, 607
vi mode, history file, 214
vi mode, in **ex**, 171
vi modes, 596
vi motion commands, 599
vi quick reference, 672
vi registers, 597
vi screen, 17
vi yank operator, 592, 604, 606
vi, see **ex** also
view file, 372, 392, 416, 589
view file, octal dump, 392
view multiple files, 372, 416, 589
view, see display
virtual memory, maximum size, 576
VISUAL variable, 356, 402
visually diplay new-lines, 188
visually diplay tabs, 188
VMS to UNIX cross reference, 663

W

wait command, 439, 614
wall clock execute time, 546, 549
wall command, 614
warn option, **ex**, 190
warning, escape w/o save, **ex**, 190
wBAUD option, **ex**, 190
wc command, 618
whatis command, 620
whence command, 622
which command, 626
which command being executed, 626
which version of shell, 9
which version of UNIX, 9
while loop, 437
while loop statement, 52
who am i command, 629
who command, 628
who is logged in, 588, 628
who string, **chmod**, 82
whoami command, 632
width, page, 425
width, printer page, 314
wild cards, see filename expansion
wild cards, see regular expressions
wildcards, disable filename, 485
wildcards, filename, 230
window option, **ex**, 190
window size, **ex**, 174, 190
word count, 618
word disgnators, **csh** history, 217
word movement, **vi**, 600
word search, **egrep**, 160
word search, **fgrep**, 223
word search, **grep**, 251
word units, **vi**, 592, 597
working directory, 11, 71, 456
wrap around buffer search, **ex**, 190
wrapmargin option, **ex**, 190
wrapscan option, **ex**, 190

write command, **ed**, 155
write command, **ex**, 180
write command, **sed**, 481
write message to user, 633
write permission, 83, 333
write permission to tty, 367, 628
write to all users, 614
write to file and standard output, 539
write to file descriptor, 266, 428
write to pipe, 428
write to user, 18
writeany option, **ex**, 190
writing programs in **nawk**, 37
writing shell scripts, 432

Y

yank command, **ex**, 176
yank operator, **vi**, 592, 604, 606
yanking text, **ex**, 176
yanking text, **vi**, 606

Z

Ctrl-Z, suspend character, 186, 523

bpb Computer & Electronics Books

bpb Computer & Electronics Books

Illustrated

UNIX

SYSTEM V

If you're looking for the complete guide to the UNIX operating system, this is it. Designed for novices as a tutorial and for experienced users as an in-depth reference, this book is written in an easy-to-follow format. Based on AT&T UNIX System V Release 4 and Berkeley Software Distribution (BSD) 4.3, this book gives readers the latest information on UNIX as well as hands-on activities that take the mystery out of the cryptic UNIX command set.

ROBERT FELPS is a software consultant specializing in the UNIX operating system. Mr. Felps' background includes classroom instruction and training for UNIX users and he has served as a UNIX System Administrator, UNIX Network Administrator, UNIX System Specialist, and a UNIX Network Product Specialist.

BPB PUBLICATIONS
B-1- Connaught Place, New Delhi-110001

Rs 180/-